Gigahorse: Thorough, Declarative Decompilation of Smart Contracts

Neville Grech
me@nevillegrech.com

Lexi Brent
lexi.brent@sydney.edu.au

Bernhard Scholz
bernhard.scholz@sydney.edu.au

Yannis Smaragdakis
smaragd@di.uoa.gr

Abstract—The rise of smart contracts—autonomous applications running on blockchains—has led to a growing number of threats, necessitating sophisticated program analysis. However, smart contracts, which transact valuable tokens and cryptocurrencies, are compiled to very low-level bytecode. It is estimated that high-level source code is publicly available for under 1% of contracts on Ethereum (the most popular smart contract blockchain).

We present the Gigahorse toolchain. At its core is a reverse compiler (i.e., a decompiler) that decompiles smart contracts from Ethereum Virtual Machine (EVM) bytecode into a high-level 3-address code representation. The new intermediate representation of smart contracts makes implicit data- and control-flow dependencies of the EVM bytecode explicit. Decompilation obviates the need for a contract’s source, and allows the analysis of both new and deployed contracts.

Gigahorse advances the state of the art on several fronts. It gives the highest analysis precision and completeness among decompilers for Ethereum smart contracts—e.g., Gigahorse can decompile over 99.98% of deployed contracts, compared to 88% for the recently-published Vandal decompiler and under 50% for the state-of-the-practice Porosity decompiler. Importantly, Gigahorse offers a full-featured toolchain for further analyses (and a “batteries included” approach, with multiple clients already implemented), together with the highest performance and scalability. Key to these improvements is Gigahorse’s use of a declarative, logic-based specification, which allows high-level insights to inform low-level decompilation.

Index Terms—Ethereum, Blockchain, Decompiler, Program Analysis, Security

I. INTRODUCTION

Distributed blockchain platforms have captured the imagination of scientists and the public alike. Blockchain technology offers decentralized consensus mechanisms for any transactions that, in the past, would have required a trusted centralized authority. One of the most evident embodiments of this vision is the development of smart contracts: Turing-complete autonomous agents that run on distributed blockchains, such as Ethereum or Cardano. A smart contract may, for instance, implement a lending policy, a charging scheme for digital goods, an auction, the full set of operations of a bank, and virtually any other logic governing multi-party transactions.

Ethereum is the best-known, most popular blockchain platform that supports full-featured smart contracts. (As of this writing, the Ethereum cryptocurrency market capitalization is $30B.) Ethereum offers an excellent demonstration of the potential for smart contracts, as well as their technical challenges. Developers typically write smart contracts in a high-level language called Solidity, which is compiled into immutable low-level Ethereum VM (EVM) bytecode for a persistent distributed virtual machine running on the Ethereum blockchain.

The open nature of smart contracts, as well as their role in handling high-value currency, raise the need for thorough contract analysis and validation. This task is hindered, however, by the low-level stack-based design of the EVM bytecode that has hardly any abstractions as found as in other languages such as Java’s virtual machine. For example, there is no notion of functions or calls. A compiler that translates to EVM bytecode needs to invent its own ABI to build stack frame for calls and calling formats.

It is telling that recent research [1], [9], [21], [33] has focused on decoding smart contracts into a higher-level representation, before applying any further (usually security-oriented) analysis mainly to expose data- and control-flow dependencies of smart contracts. Past decompilation efforts have been, at best, incomplete. The best-known decompiler (largely defining the state-of-the-practice) is Porosity [32], which in our study fails to yield results for 50% of deployed contracts of all smart contracts on the blockchain. Upcoming research tools including the Vandal decompiler [34] still fail to decompile a significant portion of real contracts (around 12%) due to the complex task of converting EVM’s stack-based operations to a register-based intermediate representation.

Such difficulties are much more than technicalities of the platform or idiosyncrasies of existing tools. Any current or future smart contract platform is likely to employ virtual machines that are low-level. The designs of these virtual machines are optimized for massively replicated executions of smart contracts. The bytecode effectively represents an assembly language that is created for efficient execution and compact program representations since the bytecode must be stored on the blockchain. Hence, the bytecode for smart contracts will never be made for human readability nor reverse compilation.

For effective decompilation, significant program comprehension of the bytecode is necessary. A decompiler requires deep program understanding before it can reconstruct the bytecode to a high-level representation. For instance, to recognize which low-level jumps correspond to high-level function calls, a decompiler must deduce potential addresses for jump instructions to be able to reconstruct the control-flow of a smart contract. Such understanding is compounding: once calls are recognized as calls (and not just as a mere intra-procedural jump), its decompilation precision can further improve, by pruning impossible targets.

In this paper, we introduce the Gigahorse toolchain for analysis of Ethereum smart contracts. Gigahorse addresses the above challenges, making the following contributions:

• It offers a highly-effective decompiler, yielding higher
precision and completeness than the state-of-the-art (e.g.,
decompling virtually all existing contracts on the Ethereum
blockchain).

- Gigahorse anchors around it a full-featured tool suite,
offering libraries for building analyses, as well as ready-
made clients for existing analyses.

- The Gigahorse approach provides new decompilation ins-
ights (possibly of value for many more platforms): As
higher level program features are discovered, they feed back
to lower level analyses. E.g., by discovering functions, stack
analyses are performed locally (more precisely), and the
effects of function calls on the stack are also summarized,
ensuring both more precise and more scalable analysis.

- Gigahorse showcases an unconventional decompilation ap-
proach, which enables the above benefits: the decompiler is
specified declaratively, using logic-based (Datalog) rules.

- Gigahorse is evaluated and its features illustrated on the
full set of smart contracts of the Ethereum blockchain.

II. BACKGROUND

We next present some background on the EVM bytecode
language and declarative static analysis.

A. Low-Level Bytecode in the Ethereum Blockchain

The EVM is a stack-based low-level intermediate represen-
tation (IR). In the bytecode form of a smart contract, symbolic
information has been replaced by numeric constants, functions
are fused together in a sea of instructions, and control-flow is
obfuscated by jump addresses that are popped from the stack.
To highlight this issue, it is instructive to compare the EVM
bytecode language to the best-known bytecode: Java (JVM)
bytecode—a much higher-level IR. The design differences
include:

- Unlike JVM bytecode, EVM does not have the notion of
structs or objects, nor does it have a concept of methods.

- Java bytecode has a rich type system, EVM bytecode has
a single type: a 256bit word.

- In JVM bytecode, stack depth is fixed under different
control-flow paths: execution cannot reach the same pro-
gram point with different stack sizes. In the EVM byte-
code, no such execution constraints exist, which make the
identification of standard control-flow constructs very hard.

- All control-flow edges (i.e., jumps) are to variables, not
constants. The destination of a jump is a value that is read
from the stack. Therefore, a value-flow analysis is necessary
even to determine the connectivity of basic blocks. In
contrast, JVM bytecode has a clearly-defined set of targets
for every jump, independent of value flow.

- JVM bytecode has defined method invocation and return
instructions. In EVM bytecode, although calls to outside a
smart contract can be resolved, function calls inside a con-
tract are translated to just jumps (to variable destinations,
per the above point). All functions of a contract are fused
in one stream of instructions, with low-level jumps as the
means to transfer control.

To call an intra-contract function, the code pushes a return
address to the stack, pushes arguments, pushes the destina-
tion block’s identifier (a hash), and performs a jump (which
pops the top stack element, to use it as a jump destination).
To return, the code pops the caller’s basic block identifier
from the stack and jumps to it.

B. Declarative Program Analysis

Our work is based on declarative static program analysis,
applied to smart contract decompilation. Declarativeness refers
to implementing an analysis as a collection of logical rules
(i.e., simple implications) that lead to inferences, which in
turn trigger more rule inferences, up to the least fixpoint. The
Datalog language is the most standard vehicle for declarative
analysis approaches, both low-level [4], [15], [18], [26], [28],
[35] and high-level [7], [11], [22], [23]. Additionally, Datalog-
style inference rules have been used in formal specification
tasks, such as the official specification of the Java VM veri-
fier [16, p.170-320].

Datalog is less of a programming language and more of
a specification language, since computation is based on two
constructs: logical implication rules, and recursion. A Datalog
rule \( \text{C}(z,x) \leftarrow \text{A}(x,y), \text{B}(y,z) \) means that if \( \text{A}(x,y) \)
and \( \text{B}(y,z) \) are both true, for some values \( x,y,z \), then \( \text{C}(z,x) \) can
be inferred. Syntactically, the left arrow symbol \( \leftarrow \) is a logical
implication symbol and separates the inferred facts (i.e.,
the body of the rule).

Recursion is the standard vehicle for expressing computa-
tions in Datalog. Static analysis tasks are typically recursive,
with multiple sub-algorithms collaborating towards a joint goal
expressing the semantics of a program. The sub-algorithms
have no clear stepwise order, but instead all refer to (yet-
incomplete) results of other sub-algorithm in a large recursive
definition. In this way, the sub-algorithms enhance each other’s
results, each benefitting the others.

The declarative nature of the Datalog language means that
any order of firing of the rules, and any order of evaluation
of a rule’s body will yield the same final result. To maintain
this property, the implication in Datalog has to be strictly
monotonic: each rule’s firing can only introduce new facts
and not hinder any of the previous inferences.

Part of the appeal of the Datalog language is that it currently
enjoys several high-performance implementations including
Soufflé [13]. Besides high-performance computation, state-of-
the-art Datalog engines offer extensions to the language ex-
pressiveness: one can relax pure declarativeness and introduce
ordering, as well as invent new values via constructor
functions. We will refer to such non-purely-declarative facilities
explicitly in our technical presentation.

III. GIGAHORSE DECOMPILATION SPECIFICATION

In this section, we present the core building blocks of the
Gigahorse decompiler. We use Datalog as a specification
language for the decompiler. The Datalog rules are simplified
but keep the essential complexity of their counterparts in
the actual Gigahorse implementation. We attempt to provide
as much technical detail as possible without sacrificing the
readability and understandability of this paper. In this effort, the exposition will be uneven: we give in-depth details in the first few subsections, and elide technical details in later parts (of both Sections III and IV) when the reader will be able to fill the gaps.

A. Overview of Decompilation Steps

We next summarize the main decompilation actions performed by the Gigahorse decompiler. For the sake of exposition, we describe a conceptual stepwise process, even though the decompiler specification is declarative and does not have an explicit order of operations. Starting from the original bytecode, the Gigahorse decompiler:

1) Finds basic block boundaries. The output to the next step is the original bytecode, split by basic blocks.
2) Performs local analysis of stack effects of basic blocks.
   The input of the next step attaches to the bytecode relevant summaries of stack effects per block.
3) Performs whole-contract context- and flow-sensitive dataflow analysis with on-the-fly control-flow graph (CFG) construction. This information is used to produce a 3-address IR, with global registers. We refer to this IR as global 3-address IR.
4) Infers function boundaries heuristically (i.e., entry and exit blocks, together with function calls) for public and private functions. The function boundaries enable the decompiler to transform the global CFG into local CFGs and a call graph.
5) Infers function arguments and return arguments for all functions, introduces fresh variables for these and performs an intra-procedural flow-sensitive analysis to infer the flow of these fresh variables. The output form is a functional 3-address IR, i.e., all variables are local variables and are scoped. Data is passed around functions through formal arguments, return arguments, or external constructs like storage and memory.

All the above steps work in concert to derive a high-level representation from the original bytecode. In this section, we focus on steps 2 and 3, and assume that the input is already parsed as statements in basic blocks (step 1). Section IV focuses on steps 4 and 5.

The original bytecode and our two IRs (the global 3-address IR and the functional 3-address IR) have common elements, but also differ in important ways. Throughout the descriptions of these, we will override certain concepts, such as statements or variables. For instance, variables in the global 3-address IR are global variables, whereas for the functional 3-address IR they are all local. The distinction between these should be clear from context.

B. Input Language

Figure 1 describes the schema of the input and main intermediate relations, together with the domains of the program representation at this level of abstraction. Stack indices \( I \) by definition are between 0 and 1023, and we assume that all arithmetic operations on \( I \) are only defined in that range (i.e., no overflow or underflows). Notice that the original bytecode relations, such as PUSH or BINOP, make no references to variables since the EVM is a stack-based machine. Relations that capture instructions refer to a unique statement identifier. The PUSH instruction pushes a constant to the top of the stack (which starts from 0). The BINOP operation denotes any binary operation that takes the first two operands from the stack and returns a result into the stack. The specific form of binary operations is elided for presentation reasons, but examples include ADD, MUL, SHA, etc.

The JUMPI operation is a no-op, which only serves to mark its statement identifier as being a valid address to jump to. The JUMPDEST operation is a conditional jump, which jumps to the address at the topmost element of the stack, if the element of stack position 1 is not zero. The input relation NEXT returns the next statement identifier in program order for a given statement identifier. The relation BLOCK maps a statement to its block, whereas BLOCKHEAD and BLOCKTAIL indicate the head and tail statements of a block. Finally, input relation PUSHESANDPOPS returns the maximum number of stack elements that a statement or an entire block pushes and pops. More precisely, for basic blocks this is a high/low-watermark computation: the maximum and negated minimum, over all points in a block, of the balance of elements pushed minus those popped at that program point.

C. Local Stack Analysis

The bottom part of Figure 1 shows the relations inferred by step 2 of the decompilation: the local stack analysis. This step summarizes the effects on the stack, per basic block, introduces the concept of variables (since none exist in the input representation), as well as performs a first value analysis, computing a static abstraction of the contents of the execution stack at every statement.

Relation LOCALDEFINES connects a statement with a freshly introduced variable, if the statement pushes new elements on the stack. Relations LOCALSTACKIN and LO-
CALStackOut model the variables (or stack aliases) that each stack position contains before and after executing each statement, respectively. For instance, LOCALStackOut(stmt, index, ψ) means that at statement stmt, stack position index contains the same value as variable ψ (freshly-introduced by some previous statement). The domain of these relations includes both variables and stack aliases (𝑉∪𝐽). A stack alias (i.e., a stack index outside the range of values pushed by the current basic block) refers to a value that existed in a given stack position before the beginning of the basic block. This is because each basic block is analyzed in isolation, and stack values can also be passed from block to block. The earlier-defined input relation PushesAndPops gives the maximum number of possible stack aliases.

Finally, relation VariableValue maps some of the variables to program constants. Any variable that is not present in this relation is considered dynamically computed, and its value is not modeled in this formalization (although we partially model some dynamic operations in the full implementation).

The rules that describe the local stack analysis are shown in Figure 2. The computation of LocalDefines introduces new variables (one per statement that needs it) via a constructor NewFreshVar, also used in later sections. This constructor function will typically be defined to retain all information passed to it:

\[ \text{NewFreshVar} \text{(stmt, 0)} = ψ, \]

\[ \text{LocalDefines} \text{(stmt, ψ)} \rightarrow \]

\[ \text{PushesAndPops} \text{(stmt, n, *)}, n > 0, !\text{DUP} \text{(stmt)}. \]

\[ \text{LocalStackOut} \text{(stmt, 0, var)} \rightarrow \]

\[ \text{LocalDefines} \text{(stmt, var)}. \]

\[ \text{LocalStackOut} \text{(stmt, 0, var)} \rightarrow \]

\[ \text{DUP} \text{(stmt)}, \text{LocalStackIn} \text{(stmt, 0, vOrI)}. \]

\[ \text{LocalStackOut} \text{(stmt, n + pushes - pops, vOrI)} \rightarrow \]

\[ \text{LocalStackIn} \text{(stmt, n, vOrI)}, \text{PushesAndPops} \text{(stmt, pushes, pops)}, n ≥ pops. \]

\[ \text{LocalStackIn} \text{(stmt, i, i)} \rightarrow \]

\[ \text{BlockHead} \text{(block, stmt)}, \text{PushesAndPops} \text{(block, *, pops)}, i < pops. \]

\[ \text{LocalStackIn} \text{(stmt, n, var)} \rightarrow \]

\[ \text{LocalStackOut} \text{(prevStmt, n, var)}, \text{Next(prevStmt, stmt)}, !\text{BlockHead} \text{(*, stmt)}. \]

\[ \text{VariableValue} \text{(var, val)} \rightarrow \]

\[ \text{PUSH} \text{(stmt, val)}, \text{LocalDefines} \text{(stmt, var).} \]

Fig. 2. Relations to perform local (within each basic block) analysis (since these are values obtained from the stack). Global stack analysis and control-flow graph (CFG) construction is one of the most computationally-heavy parts of the analysis. We introduce additional relations in Figure 3 (top) and their definition (bottom).

Relations BlockIn and BlockOut are global analogues of LocalStackIn and LocalStackOut, and are computed by transferring values between each other according to the CFG defined by GlobalCFG. Computing GlobalCFG is the main part of this analysis step. A trivial CFG edge exists (in inverse direction) between the first statement of a basic block (given by BlockHead) and any of its predecessor statements. More interestingly, a CFG edge exists between any conditional jump instruction (JUMPI) and any basic block address (i.e., a constant value) held by the variable used to denote the jump target in the program. This makes CFG construction mutually recursive with the global flow analysis, as is typical with state-of-the-art frameworks for program analysis of higher-order languages with virtual calls, such as Java [27]. The final outputs of this analysis are embedded in relations GlobalCFG and LocalUses. The latter relation represents which variable flows to which statement, in the same order as in the original EVM stack. The relation effectively connects the value flow between basic blocks, by resolving stack aliases (i.e., stack positions set by predecessor basic blocks, as defined in Section III-C) when the interconnectivity of basic blocks is determined. This again demonstrates well the compactness and power of a declarative specification.

After performing these analyses, we can now produce global 3-address code using the schema and rules listed in Figure 4. This representation is adopted from the Vandal [9, [34]
The challenge in reconstructing functions concerns private functions, which have been dissolved by the compiler. The conversion to 3-address IR at this point is straightforward. Syntax sugar and minor detail elision are employed for presentation purposes. Language syntax is quoted using [ and ] and implicitly unquoted for meta-variables. For instance, \s[to := BINOP(x, y)]\ indicates that statement \s{s} is some binary operation on \s{x} and \s{y} with its result in \s{to}, where \s{x, y, \text{and } to} are the meta-variables referring to the bytecode variables. The distinction between variables in the analyzed program and meta-variables in the IR is clear from context, therefore we simply refer to “variables”. We reuse the instruction opcodes from the original bytecode whenever these make sense. Instructions that do not define any new variables or perform computation, such as DUP or JUMPDEST are not present in this representation. From this point, unless otherwise specified, whenever we refer to “statements”, or their identifier \s{S}, we will be referring to statements in the 3-address IR representation.

IV. RECONSTRUCTING SOURCE LEVEL FUNCTIONS

An important element in the Gigahorse decompilation is the inference of functions, which have been dissolved by the compiled
code. In order to detect private functions, Gigahorse employs complex heuristics that require a full global dataflow information propagation and the construction of CFGs. The first heuristic is to look for specific instances of passing addresses on the stack inter-procedurally, when these addresses are subsequently used for further jumps after running a function. For instance, let us look at the following simplified global 3-address IR:

```
bar: ...  
    va = CONST <ret> // set return address  
    vb = CONST 0xFF // set data  
    vc = CONST <foo> // set function address  
    JUMPI vc 0x1 // jump to foo  
    ret: ...  
    JUMPI va 0x1 // jump to 'ret'
```

This program snippet first passes the return address <ret> before jumping to <foo>. At the end of <foo>, it jumps back to the return address that was passed before. This function search heuristic, therefore has to identify that (a) a basic block (return) jumps to a valid non-locally-derived address, which (b) originates at another block (the caller) that can reach the return basic block. This heuristic is condensed in a relation, which is further refined into \( \text{FnCallRet} \) by:

- Making sure that a basic block can only belong in a single function. When a conflict arises, we deterministically pick a function according to a total ordering of our choice.
- Making sure that a function can only be entered through a function call.

### C. Decomposing Functions Further

Detecting functions that return to their callers is not sufficient in the context of the EVM. Functions that terminate (e.g., halt) are disproportionately common in smart contracts, since smart contract computation is typically short. Therefore, Gigahorse employs additional logic to also detect functions that do not return. The main idea is that a basic block is in an independent function if it is reachable from two (or more) previously-identified distinct functions. This is an intuitively inevitable rule: if two functions both use the same code, this code must also be factored into a reusable fragment, i.e., a function. (Note that, unlike the technique of Section IV-B, this approach only detects source-level functions that are called more than once—functions that are called just once are inlined with no loss of precision.)

In more detail, the detection logic identifies basic blocks reachable via paths that: (a) start from two or more function entries (corresponding to previously-identified functions A and B, for instance), where (b) each path remains within its source function (A or B respectively). The process is actually recursive—as more functions are discovered, more opportunities arise for some basic block to be reachable from more functions. Although the number of functions discovered is monotonically growing, the logic is not monotonic at the Datalog syntax level: in order to describe paths that occur strictly within a single function, the logic needs to express that “a path does not cross into a different function”, i.e., to use the negation of the call-graph edges predicate, whose contents are also growing in the same computation.\(^1\) For this reason we use a fixpoint loop external to the (monotonic) Datalog rules, so that we are able to iteratively recompute relations at every step, and each relation can refer to versions at the previous iteration. (In the implementation, this is done via the standard “components” facility of the Soufflé Datalog engine [13] that Gigahorse uses.)

The complete algorithm is shown in Figure 5. The input relation, \( \text{PUBLICFunctionCall} \) is derived as described in Section IV-A, while \( \text{FnCallRet} \) is described in Section IV-B. From these two relations we compute the inputs our algorithm, i.e., \( \text{CallGraph}_0 \) and \( \text{FunctionEntry}_0 \). The algorithm then proceeds to discover new functions at each iteration (\( \text{FunctionEntry}_n \) and \( \text{CallGraph}_n \)), and each time recomputes paths reachable from the function entry (\( \text{ReachFrom}_n \)).

After computing a least fixpoint on \( \text{FunctionEntry} \), i.e.,

\[ i = i + 1. \]

\[ \text{UNTIL fixpoint(\text{FunctionEntry})} \]

Fig. 5. Heuristic for decomposing functions further.

\(^1\)No real non-monotonicity exists: even though some path may later be found to be invalid—by crossing into a different function—different paths with the same overall property inevitably exist.
no more new functions are discovered, we can proceed to compute local CFGs and call graphs. The call graphs are computed by taking a union over all \( \text{CALLGRAPH}_n \), i.e., \( \bigcup_{n \in \mathbb{N}} \text{CALLGRAPH}_n \), which we simply refer to as \( \text{CALLGRAPH} \) from this point onwards. We also assume the same for \( \text{FUNCTIONENTRY} \). The local CFGs are then computed as follows:

\[
\begin{align*}
\text{LOCALCFG}(\text{block}, \text{next}) & \leftarrow \text{GLOBALCFG}(\text{block}, \text{next}), \\
& \quad \text{!FUNCTIONENTRY}(\text{next}), \\
& \quad \text{!FNCALLRET}(*, *, \text{block}, \text{next}). \\
\end{align*}
\]

\[
\begin{align*}
\text{LOCALCFG}(\text{block}, \text{next}) & \leftarrow \text{GLOBALCFG}(\text{block}, \text{func}), \\
& \quad \text{FCALLRET}(\text{block, func, *}, \text{next}).
\end{align*}
\]

That is, the function inference helps filter out previously-inferred CFG edges, if these do not agree with the functional abstraction. Such spurious edges arise due to inherent imprecision in any static analysis. In our rules, this imprecision is mainly introduced at control-flow join points, i.e., because of multiple predecessors of a basic block getting their stack contents mixed up, based on the logic of Figure 3.

### D. Inferring Function Arguments and Local Variables

Inferring function boundaries and local CFGs is very beneficial to client analyses. Not only are local CFGs more precise (as discussed) but function-level reasoning enables summary-based analyses (which are typically very scalable and highly precise). In order to enable summary-based data-flow analyses, we need to have local variables, and minimize the number of global variables.

Gigahorse infers the number of function arguments by computing the number of caller-supplied elements that the entire function pops from the stack throughout its execution. Similarly, the number of function return arguments is inferred by calculating the balance of extra (pushed and not popped) elements pushed up to a call instruction. Unfortunately, the EVM does not guarantee that the stack depth is statically known at each program point (unlike, say, the Java VM). A good practical solution to this is to infer all possible push and pop balances along a function’s execution paths, up to a finite upper bound, and take the minimum number of these at the end of the returning basic block. This is a heuristic strategy, whose effectiveness is validated experimentally.

After inferring the number of function arguments and return values, Gigahorse proceeds to create fresh variables for them using the \textsc{NewFreshVar} constructor. This logic is elided for space reasons.

The last analysis step is to infer (non-argument) local variables and places where these variables flow to. This is a computation analogous to the introduction of variables, before functions are detected, in Section III-D (predicate \textsc{LocalUses} in Figure 4). The main output of the analysis is predicate \textsc{FunctionalUses}(s : S, i : I, v : V), which, for each statement, indicates which local variables are used and in which order. Using this relation, we can produce the final functional 3-address IR, similar to the process described in Figure 4. At this point, we can also introduce the additional opcodes \textsc{PrivateCall} and \textsc{Privatereturn}, which substitute jump instructions that call functions or return from a function.

### V. Implementation and Discussion

The specification of the previous sections captures the essence of the Gigahorse implementation. The actual Datalog specification of the decompiler has more technical details, comprising several hundred logical rules (over 3K lines of Datalog, using \textit{Soufflé} [13] as the dialect and execution engine) and a small Python scaffolding (of around 1K lines) borrowed from Vandal [34]. Compared to the specification, the full implementation contains:

- handling of the full instruction set of the EVM, as opposed to the minimal instruction set presented;
- several more decompilation heuristics, secondary but complementary to the ones discussed in Sections III and IV;
- context sensitivity throughout the rules: concepts such as \textsc{CallGraph} have their elements qualified by a “context”, which allows more precise static analysis. (We use a 1-site context—a.k.a. 1-CFA—as the default.)
- a library facilitating further client analyses;
- example clients.

We next discuss some of the above in more detail.

#### A. Output and Client Analyses

Gigahorse produces output both in structured form (i.e., tables, exported as CSV files) and in pretty-printed text. A very simple contract, below, helps as illustration.

```plaintext
function _functionSelector() {
  v27 = (CallDataLoad(0x0) / 0x1000...);
  if (0x901717d1 == v27) one();
  if (0x5fdf05d7 == v27) two();
  v27 = (CallDataLoad(0x0) / 0x1000...);
  exit();
}
function two() { f0x57(0x2); exit(); }
function one() { f0x57(0x1); exit(); }
private function f0x57(var1) {
  STORAGE[0x0] = var1; return();
}
```

Gigahorse infers four functions in total, two of which are public functions with a high-level function name. A private function, \texttt{f0x57}, is also inferred. All low-level jumps have been transformed into high-level control-flow and calls/returns. Of the jump instructions in the original contract bytecode, only the one corresponding to the return statement is polymorphic, since it can return to two callers. Gigahorse has no trouble detecting this as a proper function with return, which enables more precise data flow. The other decompilers in our evaluation set, Porosity, Vandal and EthIR, fail to find the private function and call-return pattern.

Gigahorse has been designed as a framework for writing security-related analyses on top of its high-level functional 3-address IR. In order to facilitate the development of these client analyses, Gigahorse offers additional development tools. For client analyses implemented in Datalog, we developed a bulk analyzer that can be supplied with a user-defined pipeline of client analyses. These are compiled and executed...
in parallel, in tandem with the decompilation. Client analyses for Gigahorse can be written in any language by reading the decompilation results from CSV files. In fact, one of our client analyses is a high-level pretty-printer, which takes the decompiler’s output and outputs high-level Solidity-like code, including function signatures, control-flow structures such as if statements, some types and complex expressions. This is a useful tool for the human inspection of smart contracts.

Additionally, Gigahorse offers a Datalog API for the decompilation results together with libraries of analysis functions tuned for the decompiler’s output. These libraries perform data-flow, dependency, and loop-semantic analysis. The data-flow analysis library is able to compute an intra-procedural data-flow analysis or summary-based inter-procedural data-flow analysis. The latter is enabled by the rich functional decompilation described in the previous section. All analyses are highly parametric. Users can instantiate multiple versions, each defining its own transfer functions, and can also limit the scope of an analysis (e.g., analyze only loop-induction variables within loops). The loop-semantic analysis identifies loops and other control-flow structures, their exit conditions, induction variables, dominators, etc. Using the provided libraries we have ported the recently-published MadMax client analysis for gas-related vulnerabilities [9] to the Gigahorse decompiler with similar results but with much higher performance, and for over 99.9% of the deployed smart contracts.

B. Declarativeness and Monotonicity

The declarative nature of Gigahorse has been a significant facilitator of its decompilation effectiveness. (Speculatively, this is an insight that may be also applicable in other decompilation domains, such as machine code decompilation.) Logical rules allow for concise expression of decompilation patterns and heuristics. Perhaps more importantly, separate patterns can be specified completely independently, with no ordering or other artificial dependency. Still, the independent patterns can benefit or complement each other. Dependencies between decompilation patterns (e.g., that one needs to run before another) are determined automatically by the execution engine. Dependencies between decompilation patterns arise naturally in the specification we saw in earlier sections—e.g., the call-graph informs the flow analysis and vice versa. We also saw a representative example of complementary decompilation patterns in Section IV: one pattern detects functions by tracking that they are called and return to their call site), while another detects that a block must be in a separate function because it is reachable by two existing ones. Neither pattern is more complete or more precise than the other.

The declarative nature of the rules means that the order of their application does not matter—the result will be the same regardless of the rule ordering chosen. Contradictory patterns or heuristics arise in practice, however. For instance, there may be indicators that two basic blocks belong in the same function (e.g., one is reached via a fall-through edge of a conditional jump from the other), whereas other evidence suggests the blocks need to be placed in separate functions. This is especially the case when the compilation process (usually by the Solidity compiler) has performed aggressive optimizations, such as basic block merging. Specifying the decompiler declaratively guarantees that such contradictory heuristics are automatically detected: any order-dependency in the firing of the rules is expressed as non-monotonicity in the Datalog rules (e.g., a recursive cycle that includes the negation operator over one of the recursively defined predicates), which the language implementation rejects. In such cases, it is the programmer’s responsibility to enforce ordering or rephrase the rules, as in the example of explicit layering in Figure 5.

VI. Evaluation

There are several research questions that our experiments intend to answer:

RQ1: Scalability—is Gigahorse a scalable and efficient decompiler, for all contracts in the wild?

RQ2: Completeness/Coverage—how well does the decompiled code cover the original code available in the wild?

RQ3: Precision—does the decompiled code precisely match high-level semantics?

For most of these research questions, we will be comparing Gigahorse against Porosity (the best-known, state-of-the-practice Ethereum decompiler) and Vandal [34] (a recent research tool, probably the closest comparable in the literature). In Section VI-E we also perform an experiment with less closely related tools.

Our experimental setup consists of all programs available on the Ethereum blockchain as of April 2018. This makes up the universe set of “contracts in the wild” of around 6.6 million contracts deployed from 91.8K unique programs.

We ran all systems on an idle machine with an Intel Xeon E5-2687W v4 3.00GHz and 512GB of RAM. To bound the (long) time to run experiments, we set a cutoff of 120s for decompilation. (As we show, decompilation time for successfully-decompiled contracts is on average over 10 times below this threshold, for all systems.) Any contracts that take longer to decompile are considered to timeout. All experiments were conducted in parallel. Our machine has 48 logical cores, so we ran 45 processes in parallel. Note that the Gigahorse decompiler can also be configured to parallelize decompilation within a single contract, which is an option we did not enable when analyzing multiple contracts in bulk.

We emphasize that our evaluation is quantitative, based on several metrics, but we have confirmed by manual inspection of numerous contracts that the metrics reflect well the actual quality of decompilation.

A. RQ1: Scalability

To answer this question we employ the following metrics:

Timeouts/fatal errors, i.e., proportion of contracts that were not successfully decompiled due to timeouts or fatal errors.

Contracts with at least one function detected according to the respective decompiler.

Time i.e., wall-clock time taken to compile the average contract, excluding contracts that time out.

Contract size in terms of the number of functions computed by the Gigahorse decompiler.

<table>
<thead>
<tr>
<th>Contract size</th>
<th>Compute Time</th>
<th>Timeout/Fatal Errors</th>
<th>Compilation Coverage</th>
<th>Functions Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 contract</td>
<td>20s</td>
<td>0%</td>
<td>100%</td>
<td>1</td>
</tr>
<tr>
<td>10 contracts</td>
<td>200s</td>
<td>0%</td>
<td>100%</td>
<td>10</td>
</tr>
<tr>
<td>100 contracts</td>
<td>2000s</td>
<td>0%</td>
<td>100%</td>
<td>100</td>
</tr>
</tbody>
</table>

The table above shows the results of our experiments with Gigahorse against Porosity and Vandal.
The table below summarizes the first three metrics. (We use the convention that for metrics in italics lower is better, otherwise higher is better.)

<table>
<thead>
<tr>
<th></th>
<th>unknown jump targets</th>
<th>unreachable code</th>
<th>functions / contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gigahorse</td>
<td>0.00%</td>
<td>8.7%</td>
<td>21.0</td>
</tr>
<tr>
<td>Vandal</td>
<td>0.81%</td>
<td>19.8%</td>
<td>12.2</td>
</tr>
<tr>
<td>Porosity</td>
<td>N/A</td>
<td>N/A</td>
<td>1.92</td>
</tr>
</tbody>
</table>

Gigahorse also has the highest proportion of contracts with at least a single function detected (apart from the dispatcher function). Having 0 functions is not necessarily proof of incomplete decompilation, but it is a strong indicator, especially when the numbers for Porosity and Vandal diverge from one another. Both Porosity and Vandal fail to decompile a significant portion of the contracts. As can be seen, Porosity is bimodal: it is fast for the contracts it manages to decompile, or hits a scalability wall. Our setup, giving Porosity 100x the time of its average successful decompilation (120s), still did not allow decompilation of more than half of the contracts. The contracts that are not decompiled correctly are larger in size, for both Porosity and Vandal—Figure 6 plots decompilation success vs. contract size in functions (as reported by Gigahorse, whose function inference is the most reliable).

Gigahorse also has the highest proportion of contracts with at least a single function detected (apart from the dispatcher function). Having 0 functions is not necessarily proof of incomplete decompilation, but it is a strong indicator, especially when the numbers for Porosity and Vandal diverge from Gigahorse, whose function inference is very reliable.

B. RQ2: Completeness

We employ the following metrics:

**Unknown jump targets**, i.e., proportion of jumps that do not have a target for the non-fallthrough case.

**Unreachable code**, i.e., % of unreachable basic blocks.

**Number of functions per contract**, as indicated by the respective decompiler.

The results of these metrics are shown in the following table.

<table>
<thead>
<tr>
<th></th>
<th>unknown jump targets</th>
<th>unreachable code</th>
<th>functions / contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gigahorse</td>
<td>0.00%</td>
<td>8.7%</td>
<td>21.0</td>
</tr>
<tr>
<td>Vandal</td>
<td>0.81%</td>
<td>19.8%</td>
<td>12.2</td>
</tr>
<tr>
<td>Porosity</td>
<td>N/A</td>
<td>N/A</td>
<td>1.92</td>
</tr>
</tbody>
</table>

Note that the output of Porosity is not in a format that enables the measurement of jump targets of unreachable code, so we only measure functions. Gigahorse finds jump targets for virtually all jump instructions, which translates into less unreachable code than Vandal. The number of functions detected (public or private) is also significantly higher.

C. RQ3: Precision

To answer the question we employ the following metrics:

**Polymorphic jumps**, i.e., proportion of jumps that resolve to more than one target for non-fallthrough CFG edges.

**Unstructured loops**, i.e., proportion of loops with unusual control-flow structure. This indicates imprecision in the control-flow graph.

**Loops with an exit condition**—the converse likely indicates imprecision in the control-flow graph.

**Data flows (values) per variable**, in arithmetic or assignments.

We only compare Gigahorse against Vandal in this case, as Porosity does not enable measurement of these metrics.

The detection of loops is also a client analysis offered by the Gigahorse toolchain, where we can see that there are fewer unstructured loops detected in the inferred CFGs. Since there are no high-level constructs in Solidity to write unstructured loops, we expect that a good quality CFG has virtually no unstructured loops (which is the case in Gigahorse). Here, the context sensitivity and local CFGs offered by Gigahorse directly translate into a higher-quality client analysis.

We ported the recent MadMax gas-vulnerability detector [9] (the main client of the Vandal decompiler) to Gigahorse. Gigahorse analyzes virtually all contracts, rather than 92% in the case of Vandal. The performance of the client analysis also benefits—the additional MadMax client analysis only costs 0.15s of wall clock time per contract on average. The table below summarizes the vulnerabilities flagged by MadMax with decompilation under Vandal vs. under Gigahorse. (Neither higher nor lower numbers are better by default for this table.)

<table>
<thead>
<tr>
<th></th>
<th>Unbounded Iteration</th>
<th>Overflow Loop Iter.</th>
<th>Wallet Griefing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gigahorse</td>
<td>3.9%</td>
<td>1.8%</td>
<td>0.32%</td>
</tr>
<tr>
<td>Vandal</td>
<td>4.1%</td>
<td>1.2%</td>
<td>0.12%</td>
</tr>
</tbody>
</table>
Although for liberal analyses (unbounded iteration) the improved precision of Gigahorse yields lower numbers, for more exotic vulnerability patterns the improved completeness of Gigahorse (managing to analyze some-10% more contracts, among the largest available) yields more warnings.

E. Sensitivity Analysis

We next discuss varying several dimensions of our experiments and how this affects the results.

a) Different Vandal settings: Published work on Vandal [9] claims a higher percentage of successfully decompiled contracts: 91.8% vs. 88.1% in our experiments. The numbers are close and the difference is due to slightly different settings. To confirm this, we also tried a different Vandal setup, for maximum decompilation ability, and got it to succeed in even more contracts—94.7%—but at the expense of significantly worse precision than that of our experiments.

b) Other Research Decompilers: In addition to the pure decompilers we compare against, Porosity and Vandal, several other recent research tools perform some form of decompilation before further analysis. We do not attempt a full comparison with such tools, but performed a more limited experiment of their end-to-end scalability and generality, in the same setup as for RQ1. We consider EthIR [1], Mythril [21], and Rattle [33]. (Of these, EthIR is the closest to a pure decompiler.) EthIR can handle 73% of the 91K unique contract programs, for an average time of 11.9s per contract. Most of the failing contracts produce an error-state exit. Our own inspection of the rule-based representation output confirms that EthIR only performs trivial local reasoning within each basic block. Mythril can handle 40% of 91K unique contracts with an average time of 19.1s per contract; exiting with an error state for 19% of contracts. On the other hand Rattle can handle 69.7% of the 91K unique contracts with the same 120s timeout. It produced non-timeout error-state exits on 24% of the contracts.

VII. RELATED WORK

Analysis and verification for smart contracts has received substantial attention recently due to the security issues inherent in the high-risk paradigm of smart contract development.

a) Decompilers for smart contracts: Vandal [9], [34] is an open-source framework written in Python, and consists of a decompiler, a blockchain scraper, and a set of extensible vulnerability analyses written in Soplfe [13] Datalog. Vandal’s decompiler uses a symbolic stack to convert basic blocks to a global 3-address IR, and constructs the control-flow graph incrementally. We have compared Gigahorse to Vandal extensively in previous sections. Porosity [32] is a high-level decompiler from EVM bytecode to Solidity-like source (similar to our pretty-printer output) implemented in C++. The EthIR [1] framework for high-level analysis of Ethereum bytecode based on the trace-based Oyente tool [17]. Its decompilation output introduces variables that are local to each basic block which makes the analysis trivial. Note that the EthIR framework reconstructs some high-level control and data-flow fragments from Oyente traces. Fragments of the control-flow graph that are not covered by Oyente’s traces remain undiscovered by EthIR. Mythril is a security analysis tool for Ethereum smart contracts [21]. Mythril performs decompilation aided by a symbolic execution engine (Laser-EVM). It produces traces that are used to generate an intermediate representation and it therefore suffers similar incompleteness issues as EthIR. Similarly, Rattle [33] also constructs an IR in SSA form [6] and performs program analysis on it.

b) Analysis frameworks for smart contracts: Previous work on smart contract security analysis can be classified according to its underlying techniques, including symbolic execution, formal verification, and abstract interpretation. Some of the work does not necessarily need decompilation. Systems including Oyente [17], MAIAN [24], GASPER [5] and Grossman et al.’s recent work [10] use a symbolic execution/trace semantics approach. Thus, they analyze a single execution trace only. The formal verification tool by Bhargavan et al. [3] detects vulnerabilities that include not checking the return value of external address calls, and reentrancy using F* similar techniques to convert stack-based operations to three-value semantics. We have also shown that Datalog is very well suited for describing feedback loops between different abstraction levels. We have also shown in previous sections. Porosity [32] is a high-level decompiler from EVM bytecode to Solidity-like source (similar to our pretty-printer output) implemented in C++. The EthIR [1] framework for high-level analysis of Ethereum bytecode based on the trace-based Oyente tool [17]. Its decompilation output introduces variables that are local to each basic block which makes the analysis trivial. Note that the EthIR framework reconstructs some high-level control and data-flow fragments from Oyente traces. Fragments of the control-flow

c) Decompilation for Java bytecode: There is a cornucopia of Java decompilers [8], [12], [20], [25], [31]. The Krakatoa decompiler [25] uses a pipeline of transformations to recover Java programs from Java bytecode. Gigahorse employs similar techniques to convert stack-based operations to three-address code via symbolic execution. In [8], Prolog is used to specify a Java bytecode decompiler. An evaluation paper [12] compared the performance of various Java decompilers.

VIII. CONCLUSION

We presented Gigahorse, a toolchain for decompiling and analyzing Ethereum smart contract binaries. Gigahorse leverages declarative program analysis to produce a highly-effective decompiler. The decompilation technology is unique due to the feedback loops between different abstraction levels. We have also shown that Datalog is very well suited for describing such mechanisms succinctly and without sacrificing performance. Furthermore, the Gigahorse toolchain is built as a full-fledged framework containing highly-parametric program analysis libraries and security client analyses. As part of future work we will be investigating further decompilation advances, e.g., deriving higher, source-level code or applying these decompilation techniques to other smart contract platforms. A further research avenue is the development of rich security analyses on top of the Gigahorse toolchain to identify more vulnerabilities.