Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 14-006

Conservative Signed/Unsigned Type Inference for Binaries using Minimum Cut

Qiuchen Yan and Stephen McCamant

January 29, 2014

Conservative Signed/Unsigned Type Inference for Binaries using Minimum Cut

Qiuchen Yan Stephen McCamant

University of Minnesota Department of Computer Science and Engineering

yanxx297@umn.edu, mccamant@cs.umn.edu

Abstract

Recovering variable types or other structural information from binaries is useful for reverse engineering in security, and to facilitate other kinds of analysis on binaries. However such reverse engineering tasks often lack precise problem definitions; some information is lost during compilation, and existing tools can exhibit a variety of errors. As a step in the direction of more principled reverse engineering algorithms, we isolate a sub-task of type inference, namely determining whether each integer variable is declared as signed or unsigned. The difficulty of this task arises from the fact that signedness information in a binary, when present at all, is associated with operations rather than with data locations. We propose a graph-based algorithm in which variables represent nodes and edges connect variables with the same signedness. In a program without casts or conversions, signed and unsigned variables would form distinct connected components, but when casts are present, signed and unsigned variables will be connected. Reasoning that developers prefer source code without casts, we compute a minimum cut between signed and unsigned variables, which corresponds to a minimal set of casts required for a legal typing. We evaluate this algorithm by erasing signedness information from debugging symbols, and testing how well our tool can recover it. Applying an intra-procedural version of the algorithm to the GNU Coreutils, we we observe that many variables are unconstrained as to signedness, but that it almost all cases our tool recovers either the type from the original source, or a type that yields the same program behavior.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering

General Terms Security, Verification

Keywords binary type inference, minimum s-t cut

1. Introduction

On one hand, the compilation process is clearly one that loses information: obviously comments and variable names are not reflected in the binary version of a compiled program, and as shown in an example below, there are also more subtle ways in which difference source programs can compile to the same binary program. On the other hand, compilation preserves all of the most important information about a program inasmuch as the compiled program can still execute correctly. This tension is part of why the abstract question of decompilation, whether it is possible to automatically invert the transformation performed by a compiler, has been intriguing for decades.

There are also important practical uses for recovering information from binary software. Many kinds of program analysis perform better and/or give more accurate results when applied to source code, because of its additional structure: recovering that structure would allow binary analysis to gain the same benefits. Also reverse engineering has a number of applications in software security and software engineering, such as understanding malicious software in order to defend against it, third party analysis of vulnerabilities in commercial software, and reverse engineering for interoperability. The most popular reverse-engineering tool used in security, the Hex-Rays decompiler integrated with the IDA Pro disassembler, can recognize a large number of idioms produced by common compilers, but it is a best-effort tool designed to be used by an expert analyst: it provides no guarantees that the code it generates will even compile.

It has proven difficult to construct general-purpose decompilers that translate a compiled binary back into equivalent source code. Likely one reason is that a number of different kinds of information need to be recovered to produce correct source code, and each might be useful for recovering the others. Most tools, including Hex-Rays, take a "best effort" approach in which each part of the system attempts to produce results that are most likely to reflect the original program, but there is no clear criterion for whether a particular structural result is correct or incorrect. Compared to the clearly defined concepts of soundness and behavior preservation used in forward compilers, the lack of a clear correctness criterion also makes it difficult to design and evaluate reverse engineering systems.

Given this situation, a natural direction forward is to decompose the decompilation problem into more limited forms of reverse engineering for binaries, in order to perform more principled research on these sub-problems on their own. Some work in this direction has been performed recently by the BAP group at CMU, with their TIE system for type inference [11] and recent work on control-flow structure recovery [15]. However in our view the TIE system still attempts to solve too many problems at once, leaving the correctness of its results difficult to analyze. It is a good start that the TIE paper identifies the goal of its type inference results being conservative, which is to say that it may return a range of possible result types which should always include the correct type. However the TIE system does not achieve this property in an unconditional sense, just a best-effort one. For instance the static version of TIE produces a conservative type for 93% of variables in a set of example programs. (The more recent paper [15] also identifies errors in the results of TIE as a major limitation of the combined system.)

In this work we take a further narrowing of the type-inference problem: we build a system to infer, for variables of integer type, whether they should be declared as signed or unsigned. Though this problem might initially appear much simpler than full type inference, it already exemplifies many of the reasons reverse engineering tools are difficult to design and evaluate. Signedness is not directly present in the binary the same way it is in source code, and

there may not be any single “correct” assignment of sign types for a given binary.

In this setting we propose a novel algorithm for inferring signedness types in the presence of signedness conversions that are not directly visible in the binary. (In the original source code these may have been either explicit casts or implicit conversions, usually with no difference in the binary, so our tool treats them in the same way. For brevity we sometimes use the term “casts” to refer both the explicit casts and implicit conversions.) We infer pairs of variables that are likely to have the same signedness type because there is data flow between them, and variables that are likely to be signed or unsigned based on an operation performed on them. Representing these constraints as a graph, we then find a minimum cut, a set of edges of minimal cardinality whose removal divides the graph into disconnected “signed” and “unsigned” subgraphs. The edges in the cut correspond to locations where conversions are required to make a legal typing: the choice of edges in the cut will not in general be unique, but a minimum cut corresponds to a typing with the smallest number of conversions, which is likely to correspond to programmer intent.

Given that a typing solution is not unique, it is not sufficient to check the correctness of our typing against the types found in the program source code. In fact our results appear relatively poor when judged in this way: many inferred types do not match the source code, and our tool also finds that many variables could equally well be signed or unsigned, whereas the C language forces every variable’s signedness type to be declared. However we argue that a better notion of correctness is whether the new signedness types lead to a C program whose behavior is the same as the original C program when compiled. We have not fully automated this definition, but check behavior preservation for a random sample of our tool’s results, and observe that in the vast majority of cases in which our tool infers a different type than in the original source code, changing the type would not change the program’s behavior. There are some cases in which the types inferred by our tool do not lead to the correct program behavior, but analyzing them we believe they suggest directions in which the tool can be improved in order to give results that are appropriately conservative.

The remainder of this report is organized as follows. Section 2 gives a further overview of the type inference problem we tackle and our evaluation strategies. Section 3 then describes our type inference approach in more technical detail, and Section 4 gives some implementation choices. Section 5 gives the results of our empirical experiments, and Section 6 compares our work with other research with similar techniques or goals. Finally Section 7 describes some directions for future work and Section 8 concludes.

2. Problem Overview

In this section, we give a more specific description of the signedness inference problem we address, in particular the aspects of the problem that influence our approach.

Debug information and erasure One challenge in researching reverse engineering is dependencies between recovering different kinds of program structure. It would be convenient for one reverse engineering phase to make use of information recovered by another phase, but since no high-quality open-source decompilation system is available, one might seem faced with the need to develop a complex system from scratch. However, it is not our ambition in this project to build an end-to-end decompilation system, so we take a simpler approach: we use existing information derived from the program source code as a scaffold. Our ultimate goal would be a decompilation system that requires only unadorned binary code as input. But our prototype system takes advantage of debugging information produced by a compiler, which contains the very information we are trying to recover: in particular, debugging information includes a list of the variables in a program along with their types.

```
int f_s(int x, int y,
        int z, int c) {
    int s = 0;
    int b = 1;
    int i = 0;
    for (; i != c; i++) {
        b <<= 1;
        if (x & b)
            s++;
        y -= 1;
        z *= 2;
        z |= 1;
        z = -z;
        s += z;
    }
    return s*y;
}
```

```
#define uint unsigned int
uint f_u(uint x,uint y,
         uint z,uint c) {
    uint s = 0;
    uint b = 1;
    uint i = 0;
    for (; i != c; i++) {
        b <<= 1;
        if (x & b)
            s++;
        y -= 1;
        z *= 2;
        z |= 1;
        z = -z;
        s += z;
    }
    return s*y;
}
```

Figure 1. Two C functions, one (left) using signed integers and one (right) using unsigned integers, which nonetheless have exactly the same behavior. (For instance, they generate identical binary code with Ubuntu GCC 4.6.3 in 32-bit mode.) The reason is that all of the operators used in this contrived function have the same behavior whether their arguments are signed or unsigned.

For evaluation purposes, our tool in effect erases (really, ignores) the information we are trying to recover, here the signed or unsigned status of integer variables. We can then compared our inferred results to the types in the debug information derived from the source code, for evaluation purposes. (Though as we will see later, a simple comparison is not the best evaluation.) Because of the narrow focus of our current research, we allow our tool to make broad use of other kinds of debugging information, include all type information other that signedness. This is essentially a best-case scenario in that it assumes our analysis could be a final phase: in assembling an end-to-end decompiler, one would need to choose a dependency-respecting order among recovery phases, or use an iterative approach. We leave these broader questions for future work.

Binary-invisible casts and conversions Source-code signedness types are underdetermined by a compiled binary: there will in general be a number of different assignments of signed and unsigned types that produce the same binary. This occurs because in a binary, the signed versus unsigned distinction is associated with certain operators, but not with other operators and not directly with variables (storage locations). One way in which the types are underdetermined illustrated in Figure 1: many operators represent the same bit operation on signed and unsigned values. (Here and throughout we assume that signed integers are two’s-complement, as is standard on all modern processors.) So for instance there is typically no distinction between a “signed add” and an “unsigned add” at the binary level.

Of course, the reason C has separate signed and unsigned types is that for a number of other operations, the signed and unsigned versions are different. But this distinction is based on the operations themselves, and C also allows signed values to be converted to unsigned values and vice-versa, either automatically when values of different types are used in a binary operator, or explicitly via a cast operation. Because signed and unsigned values are interchangeable at the binary level, a conversion between a signed and

```c
int64_t f_s(int c) {
    int m = 500000000;
    int sum = 3*m;
    int i;
    for (i = 1; i <= c; i++) {
        int denom = i * (2*i + 1) * (i + 1);
        sum += m / denom;
        m = -m;
    }
    return (int64_t)sum * 200000000;
}
```

```c
typedef unsigned int uint;
uint64_t f_u(uint c) {
    uint m = 500000000;
    uint sum = 3*m;
    uint i;
    for (i = 1; (int)i <= (int)c; i++) {
        uint denom = i * (2*i + 1) * (i + 1);
        sum += (int)m / (int)denom;
        m = -m;
    }
    return (int64_t)(int)sum * 200000000;
}
```

Figure 2. Two C functions, one (left) in which all of the variables are declared as signed integers, and one (right) in which they are all declared as unsigned integers, which nonetheless have exactly the same behavior. (The binaries produced by Ubuntu GCC 4.6.3 in 32-bit mode are identical except for the use of stack slots and registers.) Even though the comparison, division, and widening operators are sensitive to the difference between signed and unsigned values, the right function uses casts to invoke the same signed operations as the left function.

unsigned value of the same size will have no direct runtime effect. This leads to a more complex way in which the source types are underdetermined by the binary: the binary is determined by the types of variables and the conversions, but the conversions are not visible in the binary. This effect is illustrated with the programs of Figure 2: a program with signed types is equivalent (in behavior and binary code) to a program with unsigned types and casts.

Casts as a cut Given that there may be many possible sets of source types that correspond to the same binary, a type-inference system like ours must make a choice between them. For some applications, any legal type assignment may be equally good, but when possible, we would prefer to use types that match the choices a human program would make and maximize the readability of the generated source code. For signedness conversions, we propose that programmers prefer to minimize the number of casts and implicit conversions present in their programs: intuitively, good style suggests that variables should have the type that best matches the way they are used.

In particular, we observe that this matching between variables and uses leads to a flow-like relationship of signedness types across many variables that are connected by being used together in operations. Based on this intuition, we propose to assign types based on a graph model, where edges in the graph capture the constraints that certain variables should likely have the same signedness type. We give more details on our algorithm for constructing a graph and computing a typing based on a minimum cut in the next section.

Behavior comparison for evaluation Because there is not a unique correct assignment of signedness types given a binary, it would not be sufficient to evaluate our algorithm simply by checking whether it gives the same types that were present in the original source code. Instead, we argue that a better definition of correctness for inferred types is that they should lead to a program that has exactly the same behavior as the original program. Of course a disadvantage of the definition is that is harder to check. For now we have conservatively approximated this definition by recompiling programs with modified types and examining the resulting binaries. If two different source programs yield the same binary output (for any choice of compiler or optimization flags, assuming the compiler is correct), then those two source programs are equivalent. If two binaries are not byte-for-byte identical, we examine them by hand, to see whether we can understand the differences as being caused by rearrangement of code, differing register allocations, or other changes that do not affect behavior. If the differences are too complex to understand manually, we will conservatively not conclude that the binaries are equivalent. Because modifying the source code to change types and comparing the resulting binaries are both currently manual steps, we will perform this kind of evaluation for just a random subset of our results.

3. Approach Details

Next we give more details on our technical approach and algorithmic choices.

3.1 Machine-independent IR

Building a binary analysis tool that operates directly on a particular instruction set has two major disadvantages. First, such an analysis supports just one architecture, and must be completely rewritten to support another architecture. Second, such an analysis requires some separate code for each possible instruction: since for instance the x86 instruction set has on the order of 1000 instructions, this makes the implementation complex.

To reduce these issues, we structure our analysis instead around a machine-independent intermediate representation (IR) language as its representation of binary code. This intermediate language has a smaller number of basic constructs, so typically a single instruction will be transformed into a sequence of IR statements. However because the IR constructs are fewer in number and more orthogonal, the code for performing analysis over the IR is simpler. The IR is also not specific to a particular instruction set, which should reduce the effort needed to port the analysis to another architecture. However we have not attempted to make our system fully machine-independent yet: for instance, in some instances the algorithm matches patterns of expressions within the IR that correspond to particular x86 instructions.

3.2 Variable recovery

Variable recovery is the step of binary reverse engineering that identifies which storage locations (for instance, in registers, on the stack, or elsewhere in memory) correspond to source-level variables. Variable recovery is logically a prerequisite to type inference, since the variables are the entities that are assigned types. We avoid the need to perform variable recovery from scratch by using the variables recorded in debugging information, but our system must still perform some analysis to match variables from the debugging information with the binary operation performed upon them. We also perform SSA conversion on registers to separate distinctly-typed uses of a register as an unnamed temporary, which are not

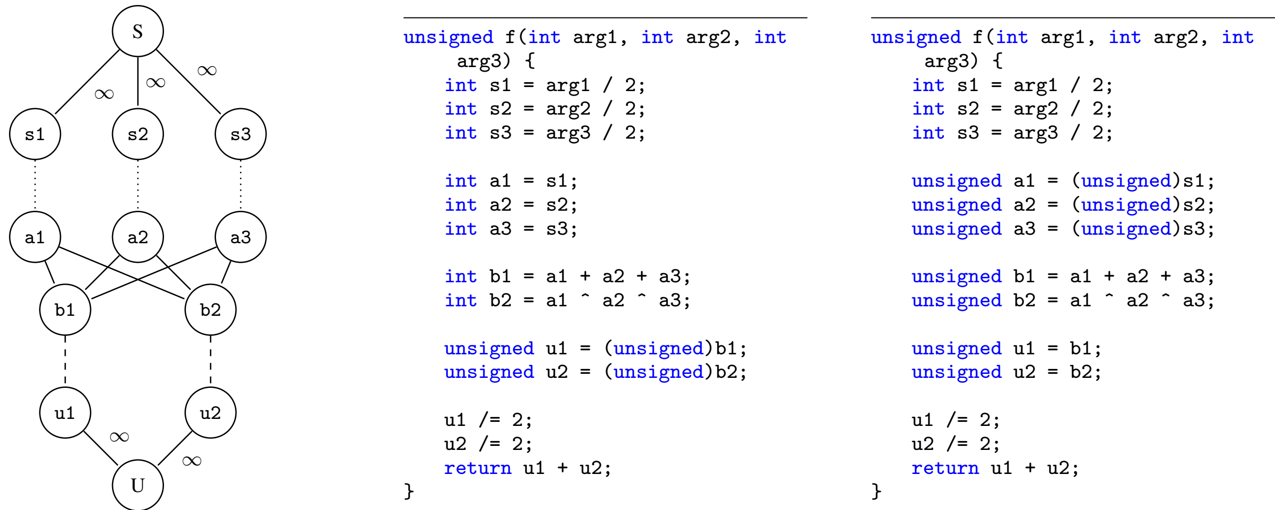

Figure 3. Example of a minimum cut. The two programs on the right are two possible interpretations of the same binary code. We can be sure that the variables `s1` through `s3` are signed, and that `u1` and `u2` are unsigned, and that a signed-to-unsigned conversion must occur somewhere in the data flow between them. But the intermediate `a` and `b` variables could be either signed or unsigned. The graph on the left shows how we use a minimum cut approach to assign types with a small number of casts. Variables known to be signed are connected with infinite-weight edges to the special node S (short for "signed"), and likewise for unsigned variables and U. All other edges have weight 1. Assigning signedness types to the variables splits the graph into two components connected to S and U; edges between the components form a cut, and correspond to casts or conversions. The dashed edges and dotted edges correspond to two different possible such cuts. Our system chooses the dashed edges, a minimum-weight cut, leading to the types shown in the left-hand code example.

recorded in the debugging information. More details of these algorithms are in Section 4.

3.3 Graph construction and cut selection

Recall that our goal is intuitively to assign signedness types in a way that minimizes the number of casts and implicit conversions required. We implement this principle by selecting types with an algorithm based on a minimum cut in a graph. Specifically, we construct a graph with a node for each integer variable in the program, and an edge connecting two nodes if those variables might be expected to have the same signedness type (such as because they were used together in an operation). We also add a distinguished node "signed" to the graph, and connect it with infinite weight edges to the nodes for variables we are confident should be signed; and likewise for "unsigned". Nodes that are connected to the "signed" node in this graph might be expected to have a signed type, and likewise for nodes connected to the "unsigned" node being unsigned. The simplest situation occurs when the graph has two connected components, one containing the "signed" node and the other containing the "unsigned" node: then the contents of these components are signed (resp. unsigned). There may also be nodes that are not connected to either "signed" or "unsigned": for these our algorithm has no basis on which to assign them either type.

However it can also occur, and in fact it often does in practice, that the "signed" and "unsigned" nodes lie in a single connected component. In this case we must divide this component in two pieces to assign types to the variables in it. An example of this situation is shown in Figure 3. More formally, this division corresponds to an *s-t cut*: a subset C of nodes in the component such that the "signed" node is in C and the "unsigned" node is in the complement of C. We can also identify the cut with the set of edges that connect a node in C to a node not in C: these are the edges that must be "cut" to separate the components. These cut edges correspond to locations in the program where a cast or implicit conversion will be required to get a correct type assignment. Since our goal is to minimize the number of such casts and conversions needed, our algorithm finds a cut such that the sum of the weights of the cut edges is minimized: a *minimum (s-t) cut* for short. We assign effectively infinite weights to the edges connecting directly to the "signed" and "unsigned" nodes, and all other edges we give a unit weight. Conveniently, our system can efficiently compute a minimum cut using an algorithm for maximum flow and the classic max-flow/min-cut duality. (We observe though that the choice of a minimum cut may still not be unique.)

3.4 Intra- vs. interprocedural analysis and libraries

We recall the standard terminology that a program analysis is *intraprocedural* if it analyzes each function in isolation, versus *interprocedural* if functions are considered together. The graph-based algorithm described above could potentially be either intra- or interprocedural: an interprocedural version would build a single large graph and add edges connecting actual arguments to formal parameters. An interprocedural analysis would be eventually preferable, since it would give more accurate results and the performance cost of computing a cut on a larger graph would probably not be prohibitive. However at the moment our implementation is intraprocedural.

A related design point is how to deal with functions whose implementation is outside the scope of analysis: for instance functions from the standard library (or if libraries are included in the program, system calls). Our analysis would be able to give more precise results if it were aware of the type constraints arising from these external functions: for instance based on debugging symbols or a manually assembled interface specification. But our current implementation treats the types of arguments to external functions as unconstrained.

4. Implementation

Our implementation can be roughly divided into two parts: variable recovery and type inference. Since variable recovery is not our main focus, we access debug information directly in order to identify each variable. Simultaneously, we translate binaries to the Vine IR, and match the debugging information for each variable to memory accesses in the Vine IR. Finally, we construct graphs which represent constraints based on the Vine IR, and solve them by computing maximum flow. Currently we have implemented a intra-procedural type inference tool: each subroutine is analyzed separately and standard (C) library functions are ignored.

4.1 Variable recovery

For this part, we use Libdwarf [2] to read debugging information. For each variable, we store its name, location, type, and length. The location and length are used (in most cases) to match a variable in the debugging information with an accessed memory block. The type is also used to decide whether two pointers have the same type.

In order to map from debugging information to memory accesses in the Vine IR, we adopt different approaches for different kinds of variables. For static or global variables, we identify the variable by its (fixed, absolute) address. For local variables, we also identify variables by their location, but the location is the sum of a register that is a pointer to the stack (`%esp` or `%ebp`) plus a constant offset.

The mapping process in the presence of pointers is more complex, since we must propagate type information across dereferences to categorize memory accessed via pointer (for instance, because it is heap-allocated). For each statement in the Vine IR, we check whether it is in the following format:

$$\textit{register1} = \textit{register0}^{?} + \textit{mem}[\textit{register2} + \textit{constant1}^{?}]$$

Items with a question mark superscript may be absent. For each statement in this format, we check whether *register2* + *constant1* is the location of a pointer variable. If the answer is yes, then *register1* contains the value of this pointer, and a subsequent access of the form *mem*[*register1* + *constant2*] represents the memory block pointed by this pointer. Note that the contents of this memory block may also be a pointer.

4.2 Type inference

To increase the accuracy of the analysis (for instance, because a single register may hold values of different types at different points), we convert the Vine IR to SSA (static single-assignment) form before doing analysis. We implemented Cooper et al.'s algorithm [5] to compute dominance frontiers. After that, we look for operations that reveal whether their operands are signed or unsigned. Based on the Vine IR and those operations, we build a graph for each function. The graph contains a "signed" node and an "unsigned" node, which are connected to the operands of signed (respectively unsigned) operations. Finally, we compute the minimum cut between the signed node and the unsigned node if they are in the same component. We regard all variables in the same component as the signed or unsigned node as signed or unsigned variables respectively.

4.2.1 Signed/Unsigned operations

The signed and unsigned operations we have found are as follows. Note that our implementation is based on operations in the Vine IR, but for simplicity of notation we give our examples in the original x86 assembly format.

- **Conditional jump**
  For each conditional jump in the Vine IR, we check its type and connect both of its operands to either the signed node or the unsigned node according to the following table, if it makes an ordered comparison.

	signed	unsigned
$>$	`jg`	`ja`
$\geq$	`jge`	`jae`
$<$	`jl`	`jb`
$\leq$	`jle`	`jbe`

- **Right shift**
  We connect a variable to the "signed" node if it is right shifted using arithmetic right shift. Otherwise, we connect this variable to the unsigned node. For example, if we see

```
sar $0x8,%eax
```

  then an edge should be added between `%eax`'s node and the "signed" node. On the other hand,

```
shr $0x8,%eax
```

  indicates that `%eax` should be connected to the "unsigned" node.

- **Modulo and divide**
  These two operations are always executed together, since `divl`/`idivl` do both of them at once. Operands of signed divide/modulo operations are connected to the signed node, while those of unsigned divide/modulo are connected to the unsigned node. For example, if we see

```
mov 0x8(%ebp),%eax
mov %eax,%edx
sar $0x1f,%edx
idivl 0xc(%ebp)
```

  then we connect `%eax` and the variable mapped to `0xc(%ebp)` to the signed node. On the other hand, those two operands are connected to the unsigned node in the following example.

```
mov 0x8(%ebp),%eax
mov $0x0,%edx
divl 0xc(%ebp)
```

4.2.2 Graph

We build a graph for each function in a program. There are three types of nodes in addition to the "signed"/"unsigned" nodes: register nodes, variable nodes and operation nodes. Both register nodes and operation nodes come from the Vine IR. Register nodes correspond to registers in SSA-form Vine IR, while variable nodes correspond to variables in the debugging information. Operation nodes also come from Vine IR. They connect operand(s) and a result together. In addition, if an expression is assigned to a variable/register, the node corresponding to this expression will be connected to the node of this variable/register. Our system uses the Boost Graph Library [16] to construct graphs.

In order to compute the minimum cut between signed and unsigned nodes, we choose the Boykov-Kolmogorov maximum flow algorithm [3] to compute the maximum flow between the "signed" and "unsigned" nodes and the capacity of each edge. Given each capacity, we travel from the "signed" node in a depth-first manner, and decrease the capacity of each traversed edge by 1. We stop if there are no reachable edges whose capacity is larger than 0. Every edge has a initial capability of 1, except edges directly connected to the "signed" or "unsigned" node. These have a effectively infinity initial capacity (currently 2^{32}) so that they are not cut.

5. Evaluation

To evaluate our tool, we tested it on 106 programs from the GNU Coreutils, and analyzed the results in detail. To begin with, we count the number of variables whose inferred type matches their

declared type. The percentage of matched variables among all test samples is given in the following below. However, matching should not be the only criterion, since two source programs can be compile to equivalent binaries even if some variables have different types. Thus we have also proposed a behavioral equivalence criterion, which we describe and then give results using sampling.

5.1 Matching rate

Averaged over all 106 GNU Coreutils, the overall matching rate is rather low, approximately 9.26%. More detail is given in the following confusion matrix. Each cell contains the number of variables whose inference result falls into one of the six conditions.

declared \ inferred	signed	unknown	unsigned
signed	911	45218	1874
unsigned	8666	46460	239

As can be seen, most non-matching variables are unknown variables, those whose nodes are neither in the signed component nor in the unsigned component. Considering that there are many short functions without signed/unsigned operations, the main cause of the high unknown rate is likely our lack of interprocedural analysis. Therefore, as described in Section 3.4, we also expect to have a much lower unknown percentage after we add interprocedural analysis to our tool.

5.2 Behavior difference

Matching rate is an imperfect evaluation criterion, so we supplement it with an evaluation based on behavior difference. Below we describe our new criterion and then give the results.

5.2.1 Definition of behavior difference

We say that two binaries (or two functions, etc.) have the same behavior if they always return the same values given an arbitrary input within their input value domain. It is natural to use program behavior as a criterion for reverse-engineering both because the program's behavior is often what we are most interested in understanding, and because behavior is what is guaranteed to be preserved by compilation, so it may be all we can recover.

A simple way to see that it can be impossible to recover exactly the original source code is to observe that two different source code programs can be compiled to exactly the same binary. Under this condition, nothing can be done to figure out which one is the "real" source code of this binary if only the binary is given. Some more complex examples involving casts appear above as Figures 1, 2, and 3, but as a small example involving only a single conversion, the two following functions, which differ just in the type for the first argument, will compile to the same binary.

```
int
func(int a,
     unsigned b)
{
       int c = a/b;
       return c;
}
```

```
int
func(unsigned a,
     unsigned b)
{
       int c = a/b;
       return c;
}
```

5.2.2 Experiment and result

Our ultimate goal would be to produce source code which when compiled gives a binary with the same behavior as the original binary. But we do not yet have a tool capable of doing this, so we start by evaluating a simpler question: for which of the integer variables in a program does the signedness type of the variable affect the program behavior? In particular we are interested about this question for variables for which our tool did not infer the same type as originally declared in the source code. If our inferred type causes the program to have different behavior, it is clearly wrong. But if changing the type does not affect behavior it may be understandable that our tool could not recover the original type.

Thus to test whether there is any behavior difference between inferred and original source code, we modify the type of a variable whose inference result does not match its original type, recompile the modified source code, and compare the behavior of the resulting binary with the original binary. Currently the approach we use for behavior comparison is to manually compare the x86 instructions. If the instructions are exactly the same, or if a different variable type only leads to superficial differences that will never change the behavior, we can conclude that this difference between the inference result and the original code is not a failure of our tool. In order to decrease complexity, we only modify the type of one variable in each experiment, and compare the function containing this variable.

Among a sample of 200 randomly selected variables for which our tool gives a non-matching result ("unknown" or different from the original source), 192 have the same behavior compared with the original binary. Thus we get 96% as an approximate proportion of variables without behavior change among all non-matching variables, with a 95% confidence interval of plus/minus 2.72%. For comparison, we also evaluated a random sample of 100 variables whose inference result matches the original type. 26% of them trigger behavior difference if modified, with a 95% confidence interval of plus/minus 8.4%.

Because our aim is to produce a conservative tool, we would like to track down the causes of all of the incorrect results produced by our system, to determine whether they result from implementation mistakes or more fundamental problems. To this end we have begun examining the 8 cases mentioned above in which our tool gives the opposite type as the original source code and the difference is behaviorally significant. As of this writing our analysis is not complete, but we have classified 4 of the behavior differences as related to signed versus unsigned comparison, 3 as related to zero-extending versus sign-extending moves, and 1 arising from sign-extension in an argument to a second function. Some of the errors potentially result from missing rules for conditional move instructions and widening conversions. In fact we have noticed that two of the erroneous results are resolved if our tool treats comparisons with a variable with a constant as revealing the signedness type of the variable. The most interesting error case results in a sign extension in a 64-bit value represented in two 32-bit registers: the compiled code copies the low word into the high word and then performs an arithmetic shift right by 31. This conceptually 64-bit value is then passed as an argument to `__umoddi3`, a special function used by GCC to implement unsigned 64-bit modulus computations. This example might be helped by idiom recognition of the double-word sign extension, or interprocedural treatment of the helper function.

6. Related Work

There are several decompilers that attempt to recover source code or human readable pseudo-source-code based on static analysis. Hex-Rays [10] is a commercial product, used alongside the well-known IDA Pro disassembler, which has seen significant adoption by security analysts. No detailed description of Hex-Rays's algorithms is available; a whitepaper [9] does not mention signedness types. No open-source tool is as practically capable as Hex-Rays; the best known open-source decompiler is probably Boomerang [8], which has a sophisticated design based on SSA [7] but has recently not been very actively maintained. Van Emmerik's thesis [7] gives a data-flow algorithm for type inference, whose

type lattice includes a distinction between signed and unsigned types.

Other systems use techniques similar to decompilation but are intended only to produce an intermediate representation for further analysis. The goal of SecondWrite [1] is to translate an input binary into a high-level version of the LLVM IR (for instance for use in binary rewriting). A recent paper [6] describes a data type detection algorithm for SecondWrite, but does not mention signedness types. Techniques based on dynamic analysis have also proved effective for security applications: REWARDS [12] and Howard [17] both focus on data structures, and have been applied in vulnerability detection and buffer overflow defense [18] respectively.

TIE [11] is another recently proposed binary type inference system, and perhaps has the most similar goals to our present work. TIE's authors stress the importance of principled design and conservative results in type inference. TIE's analysis is constraint-based and builds type constraints based on how values are used. As TIE attempts to infer types almost as general as the C type system, it requires a complex constraint language and a multi-stage constraint solving process. The evaluation in the TIE paper compares it with Hex-Rays and finds that TIE has higher accuracy and is closer to being conservative. We concur with TIE's identification of conservativeness as a design principle, but would go further in designing algorithms that should be completely conservative, so that every instance of a non-conservative result is a bug to track down and fix. TIE's complex constraint system makes it more challenging to understand the sources of errors, but it seems that that TIE's constraint generation rules are too aggressive in some circumstances. One example is that TIE has a rule inferring that the operand of a unary negation is signed: but unary negation on an unsigned value is perfectly legal in C. In analyzing one incorrect bound produced by TIE, the authors note a function that uses a signed integer to store the result of `strlen`, which is unsigned, what they refer to as a "typing error" in the original program. Though this assignment might be a bug (perhaps leading to later incorrect behavior if the string was more than 2GB long), it is completely legal as either an implicit conversion or an explicit cast in C, and it might not be a bug if another invariant ensures that the string in question is not too long.

Though it is not a complete tool in itself, the BitBlaze Vine library [19] provides an intermediate language for operating on machine code as well as a number of analysis algorithms that work on that language. In the present project we use the first stage of Vine's translation, though since we chose to implement our tool in C++ we did not make use of any of Vine's OCaml algorithms. The TIE tool is based on a similar system named BAP (Binary Analysis Platform) which is a successor to Vine.

Ours is not the first system to use maximum-flow/minimum-cut techniques in program analysis. Several systems have used maximum-flow/minimum-cut techniques for information flow analysis. Flowcheck [13] computes quantitative information flow as a maximum flow of bits of information from secret inputs to public outputs; the corresponding minimum cut is a bottleneck that captures the location of information leakage. Swift [4] is an approach to building web applications secure by construction that finds the most efficient way to divide an application into client-side and server-side computations by finding a minimum cut between the two corresponding to network communications. Somewhat further afield, Terauchi [20] uses linear programming (of which maximum flow is a special case) to infer types that show a multi-threaded program is race-free.

7. Future Work

Our experimental results so far are already encouraging, but there are several directions for future research which could improve our understanding of the problem and improve or extend our system's capabilities. We mentioned extended the analysis to be interprocedural above in Section 3.4. Some other directions:

Automating source-code experiments The first limiting factor in our behavior comparison experiments is the process of modifying program source code to have different signedness types, which is currently manual. The syntax of C type declarations is simple enough that it might be possible to find and replace integer types with simple text pattern matching. Or, another approach would be to use a library such as CIL [14] that can parse and rewrite C source code. Especially in the first case, when we would be modifying source code before the C preprocessor, the mapping from source type declarations to types in the debugging information might not be one-to-one, but at a minimum we could determine it by trial compilation.

Automating behavior comparison The other limiting factor in our current type-changing experiments is the process of comparing whether two binaries have the same behavior. Of course determining that two binaries are byte-for-byte identical can easily be done automatically, and we could extend this strategy by experimenting with different compilers or different options in the hopes of maximizing the chances we will obtain exactly identical binaries. But in other cases we clearly need to be able to automatically determine whether two related binaries have the same behavior when their instructions are similar but not the same. Checking equivalence is a complex topic on its own, but in this context we would probably concentrate on those kinds of instruction differences we observe most often in compiler output: for instance, differing stack layouts and register assignments.

Evaluating alternative cuts Another natural though inherently subjective area for evaluation is our proposal to choose signedness types corresponding to a minimum cut. How closely does this match the typing decisions made by programmers in practice? We can ask numerically how close the types chosen by developers come to our standard by measuring the number of casts and conversions they require. But it would also be interesting to examine the differences between the types chosen by developers and the types with the minimum number of casts. We could think of our system as proposing a refactoring to the existing code that changes the types of variables and reduces the total number of casts: is this refactoring desirable, for instance making the code more clear?

Comparison with other reverse-engineering tools A final standard way of evaluating our system would be to compare its results with those of similar tools. Three natural tools to compare to, as already discussed, would be the TIE system, the Boomerang decompiler, and the Hex-Rays decompiler. We chose the GNU Coreutils as test cases in part because they were also used in the TIE paper [11] to compare TIE with Hex-Rays, so we know both these tools could handle these examples as well. However because our current system represents only a subset of TIE's functionality, not much comparison can be drawn from looking just at the published results in the TIE paper. It would probably be best to test all the systems in a new unified framework. The TIE system does not appear to be available for general download, but we have not yet asked its authors whether they might make it available for this limited purpose; Hex-Rays is a commercial product available to anyone, though it is somewhat expensive.

Distinguishing pointers from integers Beyond the type inference for signedness which has been our concern in this report, another related kind of type inference that might use the same techniques is inferring whether a word value is an integer or a pointer. Like the signedness question, the pointer-integer distinction does not appear directly at the binary level: registers or memory locations

are sequences of bits that can be used in any operation. Inferring pointer versus integer types can also be seen as equivalent to deciding where to place conversions between integer types and pointer types. However the constraints that arise from data flow involving pointers and integers are different: whereas signed and unsigned values do not interact (without conversions), pointers and integers mix in addition and subtraction. This suggests that a richer kind of constraint solving than minimum cut would be required.

8. Conclusion

We have presented a new approach for determining whether integer typed variables in a binary program compiled from C source code have a signed or unsigned type. Our system determines constraints from analyzing x86 instructions transformed into a simplified IR, and constructs a graph representation in which nodes representing variables are connected if they likely have the same signedness type. In general it is not possible to recover exactly the types used in the original source, but by computing a minimum cut on the constraint graph we can find a type assignment that requires as few casts and conversions as possible. We evaluated our approach on over 100 binaries from the GNU Coreutils, including a manual analysis for 300 variables of how the behavior of the code changes if the variable is changed from signed to unsigned or vice-versa. In almost all cases, our tool infers either the type used in the original source code, or a type that gives the same program behavior.

References

[1] K. Anand, M. Smithson, A. Kotha, K. Elwazeer, and R. Barua. Decompilation to compiler high IR in a binary rewriter. `http://www.ece.umd.edu/~barua/high-IR-technical-report10.pdf`, Nov. 2010.

[2] D. Anderson. libdwarf - DWARF debugging information. `http://sourceforge.net/projects/libdwarf/`, July 2013.

[3] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(9):1124–1137, Sept. 2004. .

[4] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 31–44. ACM, 2007. ISBN 978-1-59593-591-5. . URL `http://doi.acm.org/10.1145/1294261.1294265`.

[5] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical Report TR-06-33870, Rice University Computer Science, 2001.

[6] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 51–60. ACM, 2013. ISBN 978-1-4503-2014-6. . URL `http://doi.acm.org/10.1145/2491956.2462165`.

[7] M. J. V. Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, May 2007.

[8] M. V. Emmerik, G. Krol, T. Waddington, and M. Gothe. Boomerang decompiler. `http://boomerang.sourceforge.net/`, Oct. 2012.

[9] I. Guilfanov. Decompilers and beyond. Technical report, Hex-Rays SA, 2008. `https://www.hex-rays.com/products/ida/support/ppt/decompilers_and_beyond_white_paper.pdf`.

[10] Hex-Rays SA. Hex-Rays decompiler. `https://www.hex-rays.com/products/decompiler/`, Dec. 2013.

[11] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2011.

[12] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb.–Mar. 2010.

[13] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 193–205. ACM, 2008. ISBN 978-1-59593-860-2. . URL `http://doi.acm.org/10.1145/1375581.1375606`.

[14] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction (ETAPS/CC)*, pages 213–228, Grenoble, France, Apr. 2002.

[15] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium*, Washington, DC, USA, Aug. 2013.

[16] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-72914-8.

[17] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of NDSS 2011*, San Diego, CA, 2011.

[18] A. Slowinska, T. Stancescu, and H. Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 11–11. USENIX Association, 2012. URL `http://dl.acm.org/citation.cfm?id=2342821.2342832`.

[19] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008.

[20] T. Terauchi. Checking race freedom via linear programming. In *Programming Language Design and Implementation (PLDI)*, pages 1–10, Tucson, AZ, USA, June 2008.