Software Ethology

An Accurate and Resilient Semantic Binary Analysis Framework

Derrick McKee
Purdue University

Nathan Burow
Purdue University

Mathias Payer
EPFL

ABSTRACT

When reverse engineering a binary, the analyst must first understand the semantics of the binary's functions through either manual or automatic analysis. Manual semantic analysis is time-consuming, because abstractions provided by high level languages, such as type information, variable scope, or comments are lost, and past analyses cannot apply to the current analysis task. Existing automated binary analysis tools currently suffer from low accuracy in determining semantic function identification in the presence of diverse compilation environments.

We introduce Software Ethology, a binary analysis approach for determining the semantic similarity of functions. Software Ethology abstracts semantic behavior as classification vectors of program state changes resulting from a function executing with a specified input state, and uses these vectors as a unique fingerprint for identification. All existing semantic identifiers determine function similarity via code measurements, and suffer from high inaccuracy when classifying functions from compilation environments different from their ground truth source. Since Software Ethology does not rely on code measurements, its accuracy is resilient to changes in compiler, compiler version, optimization level, or even different source implementing equivalent functionality.

Tinbergen, our prototype Software Ethology implementation, leverages a virtual execution environment and a fuzzer to generate the classification vectors. In evaluating Tinbergen's feasibility as a semantic function identifier by identifying functions in `coreutils-8.30`, we achieve a high .805 average accuracy. Compared to the state-of-the-art, Tinbergen is 1.5 orders of magnitude faster when training, 50% faster in answering queries, and, when identifying functions in binaries generated from differing compilation environments, is 30%–61% more accurate.

1 INTRODUCTION

Semantic binary analysis — the act of determining a function's use within a binary — has applications in many research and engineering areas. In software forensics, identical code in different binaries can be evidence of plagiarism or copyright violations [39]. Identical semantic code in a single binary is redundant, and can be removed by software engineers to limit binary bloat [52]. Determining how a new malware affects computer systems often starts with reverse engineering the semantic behavior of functions present in the binary [50]. Malware authors often make slight changes from existing malware to create a new strain in an effort to evade detection, and malware researchers use semantic similarity to deduce lineages in these newly discovered strains [4, 26]. More dubiously, attackers can find vulnerabilities in closed source software by performing semantic analysis on pre- and post-patched binaries [5].

As difficult as it is to determine a function's semantic behavior from source [19, 27, 28, 31, 36], it is even *more* difficult from a compiled binary [40]. To gain an understanding of what a binary does, the analyst must determine the set of changes to program state a function can conduct (e.g., what computations the functions perform, or what memory addresses are written to), ascribe a semantic meaning to each function, and then build a whole program understanding from how the semantic pieces fit together. Unfortunately, whole-program analysis remains a challenge because manual analysis is time-consuming and automated solutions [12, 16, 25, 59, 64] are inaccurate. The inaccuracy of existing automated semantic identifiers stems from the difficulty in establishing a relationship between semantic behavior and the generated code that performs the semantic behavior.

Automated solutions [54] all measure properties of binary code (e.g., order and type of instructions [59], memory locations accessed [16, 37], or number of system calls made [47]), comparing the behavior of functions based on the similarity of the implementation. These solutions assume that code similarity approximates function semantic similarity, but machine code can vary while still preserving function semantics. We propose program state change as a better, more stable semantic function identifier. Program state change as a function identifier relies on the fact that semantic behavior is stable across compilations, environments, and implementations. Thus, program state change provides an ideal fingerprint, as it is impervious to compilation environment diversity or compilation information loss. Code measurement approaches, conversely, are largely susceptible to least one of these complicating factors.

We draw an analogy between the analysis of function semantics and the biological concept of ethology (from the Greek *ethos* and *logia*, meaning the study of character). A major topic in ethology is the study of reliably predictable animal responses to the presence of known environmental stimuli. Analogously, semantically similar functions reliably make predictable changes to program state for given initial states. In this paper, we present Software Ethology, a binary semantic analysis framework. Instead of relying on measuring code properties of a function, Software Ethology abstracts functions into characteristic sets of inputs and corresponding program state changes. The core idea of Software Ethology is to observe and identify the behavior or character of functions instead of the underlying code, and then use the observed behavior as a unique function identifier. That fingerprint can then be used later

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

to identify unknown functions in a stripped binary, irrespective of the compilation environment used to generate the binary.

As a proof of concept of Software Ethology, we implement Tinbergen[1], a dynamic, fuzzer-inspired function identification tool. Tinbergen adapts mutational fuzzers to discover a subset of a function's unique set of inputs and measurable program state changes (referred to as Input/Output Vectors, or *IOVecs*). Tinbergen then organizes the discovered IOVecs into a searchable tree, from which unknown functions can be classified as a previously analyzed function, or as genuinely unique functionality, which should be the focus of manual analysis. Additionally, Tinbergen can identify possible redundant functions in binaries when two functions cannot be differentiated, or share a common parent in the tree.

We evaluate Tinbergen on accuracy amid varying compilation environments, a task existing works find difficult yet is crucial for automated reverse engineering. We measure accuracy by identifying functions in the `coreutils-8.30` application suite, and we find that Tinbergen achieves a high .805 average accuracy across 64 different compilation environments.

In short, this paper provides the following contributions:

(1) Introduction of Software Ethology, a resilient approach to semantic binary analysis.
(2) A semantic function identification tool named Tinbergen, implemented as a proof of concept of Software Ethology.
(3) An evaluation of Tinbergen on real-world applications.

2 CHALLENGES AND ASSUMPTIONS

In this section, we outline challenges for semantic function identification, and our assumptions when designing Tinbergen and Software Ethology.

2.1 Semantic Function Analysis

The manual process of reverse engineering a binary starts with extracting and disassembling the instructions that the binary contains. Due to code obfuscation techniques, getting the disassembled code can be difficult, but work on code extraction makes this feasible [32, 51]. When the code is disassembled, function identification, i.e., determining the location and size of functions, is performed. Recent work [2, 48] has shown high accuracy and generality, and many disassemblers [25, 49] provide this analysis out of the box.

Once the function bounds are identified in the code, the herculean task of semantic identification — determining what computation or program state change a function performs — begins. Semantic identification is the hardest, most time-consuming part of reverse engineering. The main culprits impeding binary semantic analysis are binary size, compilation environment diversity, and information loss during compilation. Frustratingly, these also prevent the results for manual analysis of one binary from transferring to another, i.e., each binary has to be analyzed from scratch.

As Figure 1 illustrates, even for a relatively stable API like the standard C library, binary sizes tend to grow with every subsequent version. As binaries increase in size, reverse engineering becomes more difficult, because larger sizes introduce more branches in the functions' Control-Flow Graphs (CFGs), and consequently exponentially more paths to explore for determining semantic behavior.

[1]Named for Nikolaas Tinbergen, the founder of biological ethology

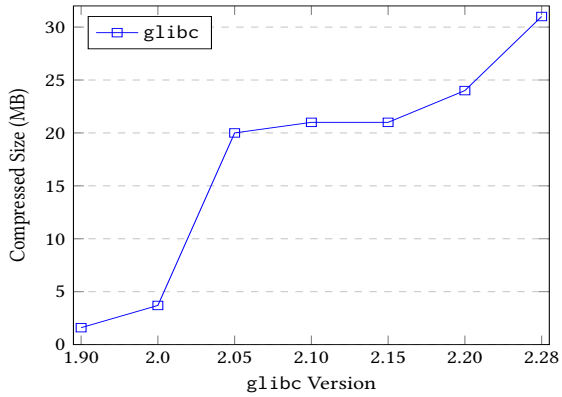

Figure 1: `glibc` source tarballs increases in size with version.

Existing semantic identifiers often perform difficult computations on code measurements, and larger code bases make such computations intractable. For example, the de facto industry standard, `BinDiff` [64], performs its binary similarity analysis by computing a graph isomorphism between two functions' CFGs, a known NP-Complete operation. Conversely, program state changes are independent of binary size creep.

The largest impediment, however, to reverse engineering a binary is the large code diversity due to different compilation environments. Here, we refer to the compilation environment as the exact compiler and linker brand and version, optimization level, compile- and link-time flags, linker scripts, underlying source, and libraries used to generate a binary.

Compilers attempt to create efficient, optimized code, and different compilers utilize different optimization sets. While compilers preserve the high level semantics expressed at the source level, the generated binary code is highly variable (see Figure 2). The source (Listing 1) is from the `strlen` implementation in the `musl` C library [43]. Different optimization levels of the same compiler (Listing 2 and Listing 3), different compilers at the same optimization level (Listing 2 and Listing 4), and different versions of the same compiler and same optimization level (Listing 2 and Listing 5) all produce different code and/or CFGs from the same underlying source. To worsen the situation, custom implementation of functions (as opposed to the use of system-distributed libraries) will likely produce significantly different binaries even in the same compilation environment. Optimizations, like dead code analysis and tail call insertions, also greatly affect the generated machine code.

Compilation removes key information from the source code that hinders manual analysis. High level features provided by the programming language (e.g., control-flow, scoping, variable types, or comments) are lost. For example, one branch in the CFG of a function might interpret a C-style `union` as a `char`, and another might interpret it as a `void*`. Regardless, in the binary, each branch would refer to the `union` in the same way, e.g., a constant offset from `rbp` for stack local variables. Such ambiguities in binaries force analysts to ascribe different semantic meanings to functions depending on the path through the CFG.

Compiler and linker differences both inadvertently conspire to prevent prior analysis from trivially transferring to new binaries.

```
size_t strlen(const char *s)
{
  const char *a = s;
  const size_t *w;
  for (; (uintptr_t)s % ALIGN; s++) if (!*s) return s-a;
  for (w = (const void *)s; !HASZERO(*w); w++);
  for (s = (const void *)w; *s; s++);
  return s-a;
}
```

Listing 1: Source

```
strlen:
  movq  %rdi, %rax
  testb $7, %dil
  je    .LBB0_4
  movq  %rdi, %rax
  .p2align  4, 0x90
.LBB0_2:
  cmpb  $0, (%rax)
  je    .LBB0_8
  addq  $1, %rax
  testb $7, %al
```

Listing 2: `clang-4.0 -O3`

```
strlen:
  pushq %rbp
  movq  %rsp, %rbp
  movq  %rdi, -16(%rbp)
  movq  -16(%rbp), %rdi
  movq  %rdi, -24(%rbp)
.LBB0_1:
  movq  -16(%rbp), %rax
  andq  $7, %rax
  cmpq  $0, %rax
  je    .LBB0_6
```

Listing 3: `clang-4.0 -O0`

```
strlen:
  testb $7, %dil
  movq  %rdi, %r8
  je    .L2
  cmpb  $0, (%rdi)
  jne   .L4
  jmp   .L22
  .p2align  4,,10
  .p2align  3
.L6:
  cmpb  $0, (%rdi)
```

Listing 4: `gcc-7.3.0 -O3`

```
strlen:
  testb $7, %dil
  movq  %rdi, %rax
  je    .LBB0_4
  movq  %rdi, %rax
  .p2align  4, 0x90
.LBB0_2:
  cmpb  $0, (%rax)
  je    .LBB0_8
  incq  %rax
  testb $7, %al
```

Listing 5: `clang-3.9 -O3`

Figure 2: `strlen` differences under varying compilation environments.

This forces the analyst to examine every function individually, even for functions that were previously analyzed. Diversity in compilation environments also presents a significant challenge to automated code-based semantic identifiers. Static analysis tools, like `BinDiff` or IDA [25], are inaccurate when presented with highly optimized code [16]. State-of-the-art dynamic analysis tools [16, 59] perform better, but still struggle with varying compilation environments or purposefully obfuscated code.

However, regardless of compilation environment, the program state changes a function performs *must* remain stable for a binary to exhibit correct behavior. Barring any bug in the compiler implementation or inconsequential actions such as dead stores, the same source code should produce the same *semantic* behavior in the final application. If this was not the case, binaries would exhibit different, and likely incorrect, behavior in different builds.

Each function implicitly specifies a unique set of input states that produce a measurable set of changes in program state, determining the behavior or character of the function. Different input states for the same function can produce different changes in program state, and different functions may produce different changes in program state for a single input state. Semantic identification can therefore be defined as the discovery of a characteristic set of input states and the corresponding external program state changes the function performs given the input. We call these inputs and program state changes Input/Output Vectors, or *IOVecs*. The challenge is to find a sufficiently distinct set of IOVecs that can uniquely identify a function. However, once found, this set can be used to identify a function in another binary regardless of compilation environment. IOVecs are thus ideal for semantic function identification.

2.2 Assumptions

When designing Software Ethology, we made the following assumptions:

(1) Binary code is stripped, but neither packed nor obfuscated.
(2) Binary code is generated from a high-level language with functions, and those function boundaries are known.
(3) Functions make state changes that are externally visible.
(4) Functions do not rely on undefined behavior.

Reverse engineers are only given the stripped binary from which to glean an understanding, and have no access to the underlying source, debugging information, symbol table, or any other human-identifiable information. This is the setting we assume for Software Ethology. We also make the assumption that all code is unpacked, and that the binary was generated from a high-level language with a notion of individual functions. The latter assumption precludes applications written wholly in assembly with no discernible functions, and, while packed code is another serious challenge in binary analysis [15, 32, 51], that topic is orthogonal to the analysis that Software Ethology performs. Finally, as it is rare in practice and most likely a bug, we assume that no code relies on undefined behavior to correctly function. Note that functions which rely on randomness are still valid (e.g., cryptographic functions); we simply assume that function semantics do not change with the compiler.

Calling a function can cause many writes to occur inside and outside the local stack frame. However, the writes a function makes cannot only be ephemeral, e.g., operating only on local stack frame memory and not returning a value. This is because once finished, any change would be overwritten or unused by later instructions, and thus the program would have been more efficient had it not called the function at all. Functions that make only ephemeral changes are dead code, and we assume that the compiler will simply remove such code from the final binary.

Note that depending on optimization level, some program state operations, e.g., dead stores which write to addresses but are never read, can be removed from the final binary. We do not include such operations in the function's set of program state changes, but focus on *persistent* and *externally measurable* program state changes. We argue that most user space functions conform to these standards, however, we discuss the limitations these standards impose in § 7.

3 SOFTWARE ETHOLOGY

Software Ethology is a function semantic identification framework, which infers program semantics by measuring the effects of execution. Instead of measuring code properties, it measures program state changes that result from executing a function with a specific initial program state.

When a function executes, it does so with registers set to specific values, and an address space in a particular state, with virtual addresses mapped or unmapped to the process' address space, and mapped addresses holding concrete values. We refer to the immediate register values and address space state as the program state. Every executed instruction does so relative to the current program state, and different paths in a function are taken depending on the initial program state.

Software Ethology performs its analysis by instantiating a specific program state before function execution, and then measuring the program state post-execution. Measurable program state changes are writes to locations pointed to by pointers, data structures, and variables whose valid lifetimes do not end when the function returns. These types of changes necessarily must be made to registers or memory addresses outside the function's stack frame. We also consider the immediate return value of a function to be a measurable program state change, but writes to general purpose registers (e.g., `rbx` on X86) and state registers (e.g., `rbp` and `rsp`) are excluded. They are excluded because, for general purpose registers, their values are immediately irrelevant upon function return, and state registers have no bearing on function semantics. Additionally, measurable program state changes preclude modifications to kernel state which cannot be reported to user space.

While executing, functions make changes to the input program state as directed by their instructions and the current program state. The same function can change the program state differently with a different input state, and different functions change the program state differently when executing with the same input state. Therefore, a function implicitly defines the input program states it *accepts* — states where the function can run and return without triggering a fatal fault — and the corresponding output program states based upon these input states. A function accordingly defines input program states it *rejects* by triggering a fault when provided a semantically invalid input state. We call these accepting input and corresponding output program states collectively Input/Output Vectors, or *IOVecs*. A function A is said to accept an IOVec I if A accepts the input program state from I, and the resulting state from executing A matches the expected program state from I. If either of these conditions do not hold, then A rejects I. See § 3.1 for the discussion of matching program states.

Given the assumption that functions make changes to input program states which are measurable post-execution, we can reframe semantic function identification. Precisely identifying a function can be seen as identifying the full set of IOVecs which a function accepts. We call that set the function's *characteristic IOVec set* (*CIS*).

```
1 int my_div(int a, int b, int* c) {
      *c = a / b;
3     return 0;
  }
```

Listing 6: An IOVec Motivating Example.

Consider the toy example in Listing 6. An accepting input program state is one that has the first argument set to any integer, the second argument set to any integer except 0, and the third argument set to any properly mapped memory address. The memory location pointed to by `c` can initially have any value. The corresponding output program state has the return value set to 0, and the memory location pointed to by `c` contains the value of `a/b`. An IOVec is a single concrete tuple of accepting input state and corresponding output state, and $CIS_{\texttt{my\_div}}$ is the full set of IOVecs `my_div` accepts. Note that only the first two arguments, the location pointed to by `c`, and the return value, are relevant, and neither the full address space nor every register value.

Every function has a *CIS*, and we hypothesize that most functions have a unique (non-empty) *CIS*. A set of functions that share a *CIS* is called an *equivalence class*. An example of an equivalence class is the set of various architecture specific implementations of `memmove` in `glibc`, e.g., `__memmove_sse2` or `__memmove_avx512`. Equivalence classes in binaries indicate duplicated functionality, and can guide engineers in reducing code bloat. For the sake of brevity, unless otherwise noted, when we refer to a function, we are actually referring to an equivalence class of functions with equal functionality.

In the general case, a function's *CIS* is unbounded. So for practical reasons, we attempt to find a subset of a function's *CIS*, which we call the *distinguishing characteristic IOVec set*, or *DCIS*. A *DCIS* for function f, $DCIS_f$, consists entirely of IOVecs which f accepts, and only f accepts every member of $DCIS_f$. Another function, g, might accept a member of $DCIS_f$, but there is at least one IOVec $I \in DCIS_f$ which g does not accept. Software Ethology is used to identify a function `foo` in a binary by providing `foo` with IOVecs $I_j \in DCIS_f$. If `foo` accepts *all* I_js, then we say that $\texttt{foo} \equiv f$.

Software Ethology needs an oracle to provide IOVecs in order to semantically identify functions, but there is no definitive source of IOVecs. Tinbergen was designed to be one such oracle, but other oracles can be devised. For example, IOVecs can be derived from unit tests or inferred from a specification. IOVecs also do not need to capture large portions of the address space in order for Software Ethology to be useful. Instead, IOVecs only need to contain the data that functions access, and oracles providing IOVecs can make any attempt at minimizing the data in an IOVec.

The number of IOVecs Software Ethology needs in order to be as precise as possible is highly dependent on the diversity and number of functions analyzed. The minimal theoretical number of needed IOVecs is equal to the number of functions being analyzed, because Software Ethology needs at least one accepting IOVec to identify and distinguish a function. However, it is likely more IOVecs are needed to precisely distinguish functions.

Currently, we utilize language semantics present in C, such as the existence of pointers and a well-defined calling convention. This is purely an implementation aspect, and different language semantics would have to be accounted for to use Software Ethology. For example, the structure of IOVecs and the oracle providing IOVecs will change when analyzing Java applications, because there is no concept of a pointer in Java. Nevertheless, the idea that program state change is a semantic identifier can be applied universally.

3.1 Matching Program States

Key to the design of Software Ethology is the definition of matching program states. Here, we present our definition of matching program states that Tinbergen uses to identify C functions.

Recall that our notion of input program state includes the arguments given to functions. Semantically similar functions should modify data contained at input pointers in similar ways (if at all), so we compare memory regions for arguments that are pointers. The target of pointers can be any arbitrary data structure, containing a mix of pointer and non-pointer data at various locations within the structure. Non-pointer values in memory regions should be bytewise the same, and any pointers to sub-objects should be located at the same offset from the base pointer. Of course, the pointer values are not expected to be the same, however, any target of pointers need to match according to these criteria as well. If there is a single mismatch in memory objects between two program states, then the states do not match.

Return values are also pertinent, but can be implementation dependent. We recognize two types of return values: pointers and non-pointers. Due to the lack of any type information in binaries, precisely determining if a return value is a pointer is challenging. We conservatively test if the return value maps to a readable region in memory, and if it does, we designate the return value as a pointer. If a return value is not readable in memory, then we consider it a non-pointer, and can represent a few classes of functions. Non-pointer return values can be the result of some direct computation on the input, such as `sin`, `abs`, or `toupper`. Non-pointer return values can also be the result of functions that assume a contract. Contract assuming functions return values that need to follow a general form detailed in a specification, but the developer is free to use any value which conforms to the specification. For example, `strcmp` can return any value < 0, $= 0$, or > 0 depending upon the memory state of the input pointers. Contract assuming functions also include functions that use return values as an indication of failure, such as `open`, which returns -1 or a valid file descriptor.

In order for two program states to match, the values contained in return registers must match in the following ways. Return values must both be pointers or non-pointers. As we do not know the size of the underlying memory region, we do not check the underlying memory values if the return values are pointers; we simply say the return values match. If the return values are non-pointers, they must be equal, or both must be positive or negative. If all input pointers (including pointers to all sub-objects) match, and the return values match, then the two program states match. As we do not perform any static analysis, `void` functions will also go through return value analysis, leading to a source of potential inaccuracy.

4 TINBERGEN DESIGN

As stated before, Software Ethology needs an oracle to provide IOVecs for semantic function identification. Ideally, a formal specification of a function's semantics would allow us to generate a minimal *DCIS*. However, such a specification is often not available, and we are forced to infer valid IOVecs. To that end, Tinbergen uses dynamic instrumentation to precisely control and monitor execution, and adapts techniques from state-of-the-art fuzzers to generate IOVecs, leveraging code coverage as an optimization and approximation of IOVec state coverage. Tinbergen can be used by analysts to identify previously analysed C functions in unknown binaries, allowing the analysts to focus on novel functionality. Additionally, Tinbergen can be used to find duplicate code when two functions cannot be distinguished when analysis is complete.

With the release of American Fuzzy Lop [62] in 2015, fuzzing became a practical means of testing large software bases, and has become an invaluable part of many large profile projects [23]. Fuzz testers, or fuzzers, come in many flavors, but mutational coverage-guided greybox fuzzers — the most widely used class of fuzzers today — are the most relevant to Tinbergen. Mutational fuzzers [3, 6, 8, 10, 21, 22, 24, 30, 38, 46, 53, 55, 60, 62] take an initial seed input, randomly permute the input, and provide the target application with the permuted input in an attempt to find new paths in the application. This process is then repeated rapidly in an effort to cover as many code paths as possible. Coverage-guided fuzzing is an efficient technique to explore as many code paths as possible without a formal specification. Different execution paths in functions tend to lead to different program state changes, which we can then measure and generate identifying IOVecs.

Due to its simplicity, fuzzing approaches quickly achieved widespread adoption. Mutational fuzzers do not require any specification of the target application; a small input file containing a few characters is often enough to generate high code path coverage. Since we have no information about an unknown function's semantic behavior, the ideas behind feedback-guided mutational fuzzing are useful in discovering IOVecs. We adopt techniques from fuzzing to generate IOVecs for use in semantic function identification, allowing us to classify functions without expert knowledge of their inner workings. By rapidly feeding a function random inputs, and measuring the program state change post-execution, we can build a corpus of function identification data without any *a priori* knowledge.

Tinbergen performs its analysis in two distinct phases: a one-time learning phase that identifies distinguishing behavior, and an identification phase for unknown binaries. In the learning phase, Tinbergen uses mutational fuzzer techniques to generate a *DCIS* for every function in a binary. Tinbergen then organizes the generated IOVecs into a binary decision tree, with each interior node representing an IOVec and a leaf representing a function. In the second phase, semantic function identification in unknown binaries consists of traversing the decision tree. Starting at the root, the unknown function is presented with the specific IOVec, and the path through the decision tree is determined by whether the function accepts the IOVec at each node. Analysis ends when a leaf is encountered, and a label is assigned to the unknown function. The following sections describe each phase in detail.

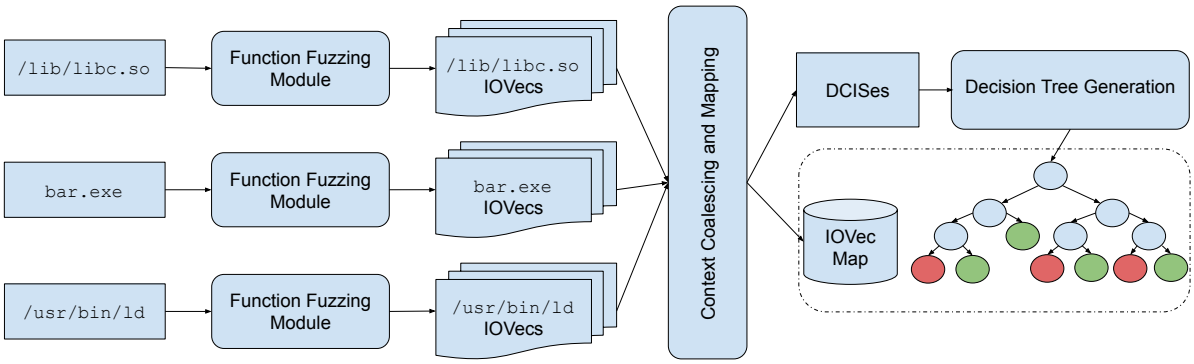

Figure 3: Tinbergen One-Time Learning Phase. The objects in the dotted box are saved for the second phase.

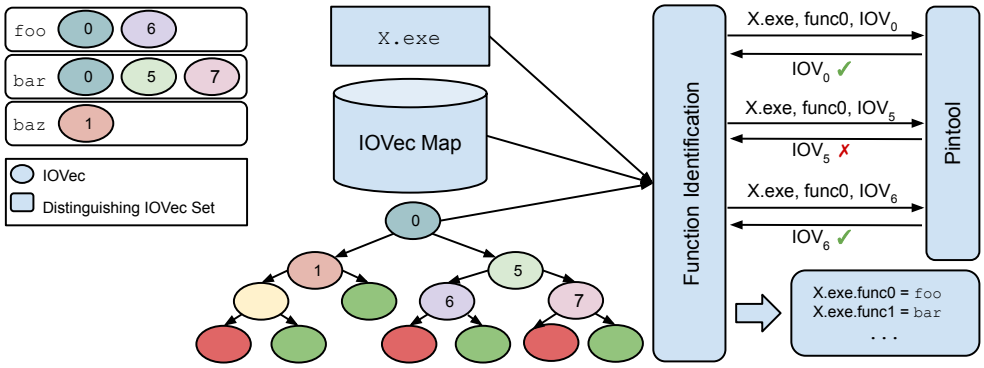

Figure 4: Tinbergen Identification Phase. The ✓ and ✗ indicates that the IOVec was accepted and rejected respectively. Paths in the tree leading to green leaves indicate semantic equivalency in the unknown binary `X.exe` to a previously analyzed function (`foo`, `bar`, or `baz`), while paths leading to red leaves represent unseen/new behavior.

4.1 Learning Phase

Figure 3 shows the overall design of the first phase of Tinbergen's binary analysis. Tinbergen supports analyzing any executable code, including shared libraries. Static libraries need to be included in either a shared library or executable. Note that Tinbergen does not require the source or any debug information. However, it does need the starting location of each function in an executable, or the exported symbol names in a shared library. Once provided with this information [2, 48], Tinbergen then enters its exploration phase.

Tinbergen employs fuzzing as a way to discover valid IOVecs. We chose fuzzing as our exploration strategy because fuzzing is optimized to maximize code coverage, leading to maximal program state change coverage, which we use for identification. Of course, we are not restricted to using fuzzing to provide an IOVec oracle, and Tinbergen can be extended to incorporate any IOVec generator.

For each Function Under Test (FUT), Tinbergen fuzzes the input arguments (see § 5), and then begins executing the FUT with this randomized program state. If that program state is accepted (see § 3) then the IOVec consisting of the initial input state and the state immediately upon return is added to the FUT's *DCIS*. After an IOVec has been successfully generated, the input arguments are again fuzzed, and the FUT is executed again with the new input program state. This process continues for a user specified amount of time. The exact amount of time the fuzzing process takes is not very critical. If the initial fuzzing stage does not provide enough distinguishing IOVecs, then it can be repeated for the functions with low quality IOVecs that do not trigger differentiable program state change. While the fuzzing process does not result in a provably minimal FUT *DCIS*, the generated *DCIS* is still compact due to our use of coverage-guided fuzzing, which is optimized to achieve high code coverage, and accordingly, high IOVec state coverage.

As mentioned earlier, we only need to save the data we expect to change. Every saved input state contains the input arguments to the FUT, and if an input argument is a pointer, the contents of the allocated memory regions. See § 5.1 for discussion on how an input argument is upgraded to a pointer. The saved output program state consists of the contents of allocated memory regions, a boolean indicating whether the return value is a pointer (see § 3.1), and the concrete value if the return value is not a pointer. We do not attempt to capture the entire address space in an effort to minimize the size of the captured IOVecs.

Before every instruction a FUT executes, Tinbergen records the immediate values of the general purpose registers, the PC register, and the stack and base pointer registers. These are used for taint analysis if a segmentation fault occurs (see § 5.1), and are discarded after every FUT execution iteration.

Once all initial *DCISes* are generated, Tinbergen enters its coalescing stage. A hash of the non-pointer register values and non-pointer memory values in allocated regions is generated for every IOVec, and a file (the IOVec Map in Figure 3) mapping hashes to IOVecs is saved for the identification stage. Because at this point in the analysis no $DCIS_f$ contains the full ground truth, a "*cross pollination process*" must occur. This is to ensure that when the decision tree is generated later, a proper ordering can be achieved, and a decision at each interior node does not lead to an incorrect conclusion. Therefore, each function for which a *DCIS* has been created is given every IOVec in the IOVec Map, and one of four results can occur for each IOVec test:

(1) The function receives a fatal signal (e.g., `SEGSIGV`), due to an improper input program state.
(2) The function does not finish executing before a user-specified time expires.
(3) The function returns, but the final program state does not match the expected output program state.
(4) The function returns, and the final program matches the expected output program state.

If and only if the last event occurs, then the IOVec is added to $DCIS_f$. As future work, we want to incorporate rejected IOVecs into the identification process as rejected IOVecs classify the *excluded* semantics of this function. All *DCISes* are then given to a decision tree generator. The resulting decision tree consists of singular IOVec hashes as the interior nodes, and functions at the leaves. This decision tree is also saved for the identification phase. Every path in the decision tree enumerates the minimal $DCIS_f$ needed to distinguish f from any other function.

If the same path in the decision tree maps to more than one function, then a potential equivalence class exists in the binary. The functions in the leaf are those for which the generated *DCIS* is insufficient to fully distinguish one function from another. This can be because of poor functionality coverage during the fuzzing stage, or because the members are truly an equivalence class.

4.2 Identification Phase

Figure 4 illustrates the design of the identification phase of Tinbergen. The analyst provides Tinbergen with an unknown binary, and functions in the unknown binary are given a label based on which IOVecs are accepted or rejected. The decision tree only needs to be created once during the learning phase, and then can be used to identify functions in any number of binaries, without any prior knowledge of the compilation environment. This is because Tinbergen, and Software Ethology in general, does not rely on code measurements, but instead relies on program state measurements for semantic identification.

Recall that the interior nodes of the decision tree consists of IOVec hashes. Starting from the root of the decision tree, Tinbergen looks up the IOVec in the map using the hash, and passes the IOVec to the unknown function. If the IOVec is accepted (see § 3), the *true* branch (right path) in the decision tree is taken; otherwise, the *false* branch (left path) is taken. The unknown function is then tested against another IOVec depending on the path taken. When the path arrives at a leaf, the unknown function is tested against one more IOVec from the leaf function's *DCIS* for confirmation. Again, if the IOVec is accepted, then the function is given the label of the function at the leaf. If the unknown function gets to a leaf and remains unconfirmed, then the function is labeled as unknown.

5 TINBERGEN IMPLEMENTATION

Here we will discuss details of our implementation of Tinbergen, and how input arguments are upgraded to pointers during the learning phase (see § 4.1). Currently, our implementation focuses on 64-bit Linux executables and shared libraries derived from C source code. Our implementation uses the Intel Pin 3.7 [11] binary translator, with 2.493 lines of C++ code, and 1.600 lines of Python. Pin readily supports Windows and 32-bit systems, adding support for these systems would require minor engineering efforts.

Fuzzing Strategy. Previous work on fuzzing has lead to the discovery of best practices for generating inputs that are most likely to maximize code coverage. Determining a function's *CIS* can be seen as a proxy for maximal code coverage. We, therefore, adopt the same best practices when fuzzing a function's input. Instead of purely random values, recent work [60] shows that preferring a specific group of input values (the *interesting values* in Listing 7), such as 0, −1, and `INT_MAX`, produce higher coverage. As a result, Tinbergen prefers setting register values or allocated areas with those values. However, as Listing 7 shows, we also employ other standard practice operations, such as random bit flips or arithmetic operations on random locations.

In order to discover a function's *DCIS*, Tinbergen fuzzes the input arguments of a function, and independently executes the function with that fuzzed program state. We currently have not implemented special support to detect arguments passed on the stack (e.g., for variadic functions or arguments passed by value). Tinbergen fuzzes the six argument registers, and memory regions for arguments which are pointers, which implicitly assumes that every function has the same arity. This is sufficient as most functions have an arity less than six, and any "used" argument register can be safely overwritten by the lower arity function.

5.1 Pointer Derivation

A major challenge to fuzzing functions (although not a general problem with Software Ethology) is the detection of pointers as input arguments. Because type information is missing in binaries, determining if an input argument is a pointer is an ongoing research topic [41, 44]. Without recovering which arguments are pointers, determining a *DCIS* for a function is generally impossible, and only incomplete behavior will be captured.

A naïve solution would be to simply replace an invalid address with a valid address before an illegal dereference occurs. While such a solution has been successfully used to solve other problems in binary analysis [29, 45], it would not work in Software Ethology, because the underlying problem — semantically, an input is supposed to be a pointer when it is not — remains unsolved. Software Ethology relies on measuring program state changes that arise from executing a function with a specific input program state. By replacing an illegal address *in situ*, the resulting output program state does not necessarily arise from actions the functions performs given the initial state.

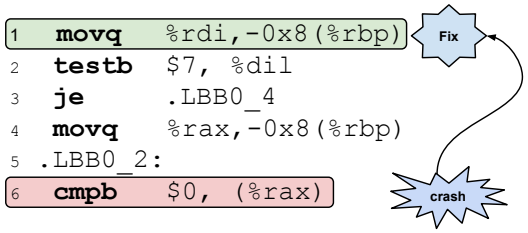

Figure 5: Backwards taint analysis to fix pointer arguments.

Consider the code in Figure 5, which is adapted from the `strlen` assembly in Figure 2. The first pointer argument to `strlen` (passed in using register `rdi`) is stored on the stack (line 1). Later, that address is written to register `rax` (line 4), and then is dereferenced and compared with the null terminator (line 6). Our fuzzing strategy is unlikely to supply a valid address as input, and line 6 will cause a `SEGSIGV` signal to be issued.

The naïve approach would replace the invalid address in `rax` with a valid address. If the function later returns with no other issue, then Tinbergen would register `strlen` as accepting the input program state with `rdi` set to a random (non-pointer) value. This is incorrect, and during the identification phase, an implementation of `strlen` in an unknown binary would *not* accept the input program state. That `strlen` implementation would then be marked as an unknown function, or worse, as a different, incorrect function.

The solution we propose is a backwards taint analysis inspired by Wang et al. [59]. As mentioned in § 4.1, while generating IOVecs in Phase I, Tinbergen records immediate register values before every instruction executes, and, if a segmentation fault occurs, we get the faulting register through a Pin API function[2]. The resulting register is the taint source, and Tinbergen then uses the saved register values to propagate the taint back to a root sink. The taint propagation policy is listed in Figure 6. Starting from the last executed instruction, each instruction is parsed in reverse order until all instructions are iterated through. The root sink is the last tainted register after all instructions are processed. We currently only support upgrading function arguments, and not global variables.

When a register sink is discovered, Tinbergen allocates a fixed-size memory region, and begins executing the FUT from its beginning with the sink register set to the address of the allocated region. If another segmentation fault occurs due to an argument which is already a pointer, then the underlying object is expected to contain another pointer. In this case, another fixed-size memory region is allocated, and its address is written to the parent memory region at the appropriate location based on the faulting address.

This process continues until the FUT successfully completes, at which point Tinbergen records the correctly initialized input program state, and the corresponding output program state, along with the coarse-grained object structure derived from the backwards taint analysis. Tinbergen only tracks which memory areas are supposed to be pointers, and no other semantic meaning is given to memory regions containing non-pointer data. Further fuzzing iterations maintain the memory object structure, and only the non-pointer memory areas are fuzzed.

[2]In Pin 3.7, we used `INS_OperandMemoryBaseReg`

Policy	Instruction	t Tainted?	u Tainted?	Taint Policy
1	$t = u$	Yes	No	T(u); R(t)
2	$t = u$	No	Yes	
3	$t = u$	Yes	Yes	
4	$t = t \circ u$	Any	Any	

Figure 6: Backwards Taint Propagation. t and u can be a register or memory address. T(x) taints x and R(x) removes taint from x. $\circ$ denotes any logic or arithmetic operator.

6 EVALUATION

We performed our evaluation using an Intel Core i7-4790 CPU, with 16 GB of RAM, and running Ubuntu 18.04.1 LTS. In our evaluation, we address the following research questions:

(1) How accurate is Software Ethology in identifying functions in binaries?
(2) Is Software Ethology truly resilient against compilation environment changes?
(3) Can Software Ethology correctly semantically identify functions missing in previous analyses?

Our results do in fact show that Software Ethology is a feasible and accurate semantic function identifier. Additionally, our results show that Software Ethology is largely unaffected by compilation environment changes, and that Software Ethology can quickly identify previously analyzed functions in novel binaries.

6.1 Accuracy Amid Environment Changes

In order to evaluate Tinbergen's accuracy, we identify functions in the `coreutils-8.30` application suite. We compiled the set of applications using `gcc 7.3.0` [56] and `clang 6.0.0` [34], at `O0`–`O3` optimization levels, resulting in a set of 844 binaries for which we semantically identify functions. Additionally, each executable was statically linked to provide the broadest possible unknown function set. In order to establish ground truth, we compiled all binaries with debug symbols enabled (`-g`). However, Tinbergen does not use them for any of its analyses, and they were only used for determining accuracy after all analyses had completed.

The accuracy numbers we report are the ratio of correctly given labels to all assigned labels, or $(TP+TN)/(TP+FP+TN+FN)$, with TP as True Positive, FP as False Positive, TN as True Negative, and FN as False Negative. A true positive is when the FUT is assigned an equivalence class in which the function is classified, and a true negative is the assignment of unknown to a function that is not present in the decision tree. The mislabeling of a function, either by assigning the incorrect equivalence class or labeling a known function as unknown, incurs a false positive and false negative penalty. The false positive comes from the incorrect label given to the function, and the false negative comes from our claim that the function does not belong to its correct equivalence class.

To conduct our evaluation, we built a decision tree (see § 4.1) using a random binary from the evaluation suite generated by each compiler and each optimization level (for a total of 8 decision trees). We chose random binaries to demonstrate the fact that Tinbergen is consistently accurate, regardless of how a decision tree was

D-Tree \ Suite		O0		O1		O2		O3	
		gcc	LLVM	gcc	LLVM	gcc	LLVM	gcc	LLVM
O0	gcc	**.863**	.850	.854	.800	.822	.841	.850	.840
	LLVM	.810	**.818**	.823	.770	.787	.811	.813	.810
O1	gcc	.798	.803	**.854**	.783	.806	.832	.836	.834
	LLVM	.757	.756	.781	**.811**	.795	.806	.788	.806
O2	gcc	.757	.758	.789	.774	**.833**	.800	.806	.801
	LLVM	.759	.769	.799	.774	.786	**.827**	.794	.826
O3	gcc	.783	.781	.814	.767	.804	.812	**.839**	.814
	LLVM	.770	.775	.806	.779	.800	.831	.811	**.833**

Figure 7: Average accuracy for `coreutils-8.30` per decision tree compilation environment (rows) across evaluation suite compilation environments.

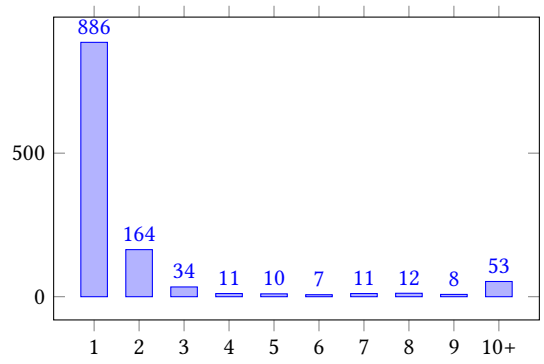

Figure 8: Distribution of all equivalence class sizes across all decision trees in the `coreutils-8.30` evaluation.

generated; Tinbergen should not rely upon choosing an "optimal" binary to maximize accuracy. For every decision tree, we used that tree to identify functions in all 844 available binaries in the evaluation suite. In order to determine the correctness of a label, we performed a simple string comparison between the name of the Function Under Test and the functions in the assigned equivalence class. If any matched, we record a true positive, otherwise we record a mislabel. Likewise, if a function is labeled as unknown, and its name appears in any equivalence class, a mislabel is assigned. Otherwise, a true negative is recorded. As the number of classified functions is large and consistent for each binary used (see Figure 9), we believe that our experimental setup is a fair evaluation, as well as indicative as to how we expect Tinbergen to be used in practice.

Figure 7 shows the average accuracy achieved with each decision tree. Each row reports the accuracy of the specific decision tree has when used to identify functions in binaries generated with a specific compilation environment (presented as the columns). The diagonal numbers (in bold) are, therefore, the accuracy rates when the decision tree and evaluation suite match in both compiler and optimization level. They are unsurprisingly among the most accurate Tinbergen achieved, and represent the data most reported by other related works. Overall, we achieve a high .805 accuracy rate, but the diagonal numbers — the numbers which allows for the best apples-to-apples comparison with related works — is .835. We believe that these results show that Software Ethology is accurate as a semantic function identifier, as well as largely resilient to compilation environments (Research Questions 1 and 2).

Unfortunately, the two closest systems to Tinbergen, `BLEX` [16] and `IMF-SIM` [59] are closed source, and we could neither obtain the source code nor the detailed results from the authors. However, our results are similar, if not better, than the results reported by the authors in the respective papers. The `BLEX` authors report an accuracy of .50–.64 across three compilers (they added Intel's `icc` compiler) and four optimization levels, and the `IMF-SIM` authors report an accuracy of .57–.66 across three compilers and three optimization levels. Both systems attempt to build a classification vector from code measurements, and their lowest inaccuracies come from labeling functions in binaries from compilation environments different from their source models. Tinbergen, in contrast, is accurate regardless of compilation environment, as evidenced by the off-diagonal numbers in Figure 7. Our results show an average 30%–61% increase in accuracy in differing compilation environments over these competing works. Note that we will release our Tinbergen prototype along with our full results as open-source with the published paper.

We identify two major sources of inaccuracy: false mislabel designations, and state dependence. We give concrete examples of both in Figure 11, which list five of the most commonly mislabeled functions in our tests. False mislabel designations are the assignment of a mislabel to a function, when the label is in fact correct — a key feature of Software Ethology that artificially lowers our accuracy measurements. State dependence mislabels are the result of missing some needed state for proper classification.

False Mislabels. Functions present in less optimized binaries might be missing from more highly optimized binaries. The compiler, for instance, might opt to replace a wrapper function with the underlying function. This behavior, while reasonable for performance, makes determining our true accuracy difficult, as a precise ground truth requires manual source analysis. For example, `xcharalloc` (if missing from the decision tree) is always equated with the function `xmalloc`, which is correct because they differ only by return pointer type. However, that determination can only be made by examining the source. As our *evaluation metric* relies upon a full string match between the FUT name and the assigned equivalence class, a (false) mislabel is given. As another example, the function `c_isalpha` (and its family) is given the false mislabel `__iswalpha` (or related), despite their equivalence in a default system environment. We ensured that the hardware-specific memory comparison and copying functions (e.g., `memcpy` and `strcmp`) are properly equated, as they are interchangeable. Generally, though, without expert knowledge or extensive source analysis, true equivalences cannot be known, and we opted to be as conservative as possible when determining correctness by performing no further analysis. Note that the inaccuracy introduced through false mislabels is due to our *conservative evaluation metric* and lack of precise ground truth, and not due to an inherent flaw in Tinbergen or Software Ethology. In fact, false mislabels give evidence that we satisfy Research Question 3, but due to the inherent imprecision of our evaluation metric, the presented results are an *under-approximation*

	O0		O1		O2		O3	
	gcc	LLVM	gcc	LLVM	gcc	LLVM	gcc	LLVM
N	614	442	461	458	438	420	584	434
$\overline{N}$	2.94	2.78	2.78	2.83	2.83	2.82	2.89	2 .84

Figure 9: Total number of classified functions (N), and average number of functions per equivalence class ($\overline{N}$), per generated decision tree in the `coreutils-8.30` evaluation. The median equivalence class size is 1.00 for all 8 decision trees.

of the accuracy. Ideally, we would require a fully precise ground truth but this would require extensive manual labeling.

State Dependence. For simplicity, we designed Tinbergen to assume nothing when generating IOVecs, and it always executes functions in isolation. However, there are functions (e.g. `close` and `munmap`) that depend on the results of previous functions in order for the input arguments to be semantically correct. `close` requires that the input integer be a valid open file descriptor (as obtained from `open`), and any input that is *not* a valid file descriptor is semantically incorrect. Similarly, `munmap` requires a valid pointer obtained via a previous call to `mmap`. Because Tinbergen does not perform any initial setup to obtain semantically correct input values, these functions are likely to be misclassified. This is not a fundamental flaw in Software Ethology, but instead is a result of Tinbergen's focus on user-space functions. We expect that our accuracy would improve significantly if we added some common environmental activities (such as opening file descriptors or memory mapping address spaces) to our IOVec design. We have it as future work to incorporate application specific environmental setup to Tinbergen.

6.2 Equivalence Class Distributions

Figure 9 shows the total number of classified functions (N), and the average number of functions per equivalence class ($\overline{N}$). Ideally, $\overline{N}$ should be close to one, as most functions provide unique and singular functionality, and thus should be assigned as the sole member of an unique equivalence class. However, with the existence of wrapper functions and hardware-specific implementations of the same function, it is likely $\overline{N}$ will be higher. It nevertheless should be low, because one could trivially get high accuracy by grouping all functions into the same equivalence class. As Figure 9 shows, we achieve a low $\overline{N}$ across our decision trees, which gives an indication that our fuzzing strategy is a generally sound technique for generating sufficiently distinctive IOVecs. We also achieve a high N across all decision trees, demonstrating the universality of Tinbergen, since we are able to classify a large number of diverse functions. Additionally, the equivalence class size distribution shown in Figure 8 (with a more granular breakdown provided in Figure 12 in the appendix) shows that we are creating hundreds of equivalence classes with one or two functions per equivalence class. We, therefore, claim that our accuracy comes from Tinbergen's ability to distinguish function semantics, and is not simply grouping all functions into a few equivalence classes.

For each decision tree, there are some equivalence classes containing a large (more than 10) number of functions. These are cases where our fuzzing strategy was unable to trigger deep functionality, yet the classified functions share a common failure mode (e.g. return −1 for invalid input). Improvements in related fuzzing work will directly translate to an improvement of coverage and a reduction of the size of these equivalence classes. Additionally, approximately 15% of the functions in the large equivalence classes are functions that are used to provide runtime dispatch of different C library functions[3]. These functions take in no argument, perform some analysis on the hardware currently present, and then return the pointer of the specific implementation to use. As these functions violate our assumption that functions change program state when executed, Software Ethology will struggle to identify such functions. However, these functions are designed to never be called directly, and are purely an artifact of using `glibc`.

6.3 Training and Labeling Time

On average, Tinbergen takes 16 CPU hours to generate a decision tree, which includes generating IOVecs and the cross pollination phase described in § 4.1. As stated before, however, this analysis only needs to be done once. Once the decision tree is generated, semantic analysis is very quick, taking, on average, only 14 CPU minutes to classify a binary in the evaluation set. Note that all operations in both of Tinbergen's phases represent completely independent work loads, and as such are embarrassingly parallel. Therefore, execution time varies with the available hardware.

`BLEX` reports 1,368 CPU hours for training, and 30 CPU minutes to classify a binary in coreutils. `IMF-SIM` takes 1,027 CPU hours for training, and 31 CPU minutes to classify a coreutils binary. Although there are hardware differences between the related works and our experimental setup, we are confident that Tinbergen's training is at least 1.5 orders of magnitude faster, and 50% faster at classifying binaries. We believe that we are faster at semantic queries because we organize past analysis in a tree-like structure; `BLEX` and `IMF-SIM` must compare the feature vector they record with every past feature vector, creating an $O(log(n))$ vs. $O(n)$ search performance disparity.

6.4 Imprecision and IOVec Case Studies

Large Equivalence Classes. To provide a concrete example of an equivalence class, as well as illustrate the issues with evaluation, we want to highlight a specific large equivalence class generated for the `LLVM O2` decision tree. The functions comprising this equivalence class are listed in Figure 10, and include several hardware specific implementations of `memcmp`, `memchr`, and `strncmp`, as well as several variants of functions which write characters to a file descriptor (the `_IO*` functions).

In our evaluation, we would record a true positive if, for example, `__memchr_sse2` from a binary was assigned this equivalence class, because that function name is present in the equivalence class. However, we would record a false positive and false negative if `my_memchr`, a different, semantically equivalent implementation of the same function, was assigned this equivalence class. This is because the function name is not present in the list of functions of the equivalence class. Obviously, this function belongs to this equivalence class, but we conservatively say we have mislabeled in

[3]The so-called Indirect Functions in `glibc`, which are different from function pointers.

_IO_default_xsputn	_IO_do_write
_IO_file_xsputn	_IO_wdefault_xsputn
__memchr_sse2	__memrchr_avx2
__memcmp_avx2_movbe	__memcmp_sse2
__memcmp_sse4_1	__memcmp_ssse3
__strncmp_sse2	__strncmp_ssse3

Figure 10: An equivalence class in the `LLVM O2` decision tree.

order to minimize possible bias that might influence our reported accuracy. For this reason, we claim that our reported accuracy is the *lower* bound that Tinbergen can achieve, and that our true accuracy is significantly higher.

The equivalence class also highlights the effectiveness and limitations of our fuzzing strategy and program state comparison policy. The functions in Figure 10 all share similar signatures, in that they have a pointer as the first argument, and the second argument is either a integer or a pointer (which is also a valid integer). However, the functions in the equivalence class are not the same, and our fuzzing round was not good enough to distinguish clear differences in functionality. Any improvement in Tinbergen's fuzzing engine (e.g., supplying better seeds or using more sophisticated guided fuzzing) would likely be enough to better differentiate the classes of functions. For example, the reason why the IO functions are grouped in this equivalence class is because the input `FILE*` object is not valid, and they are exiting early by returning -1. This resulting program state is exactly the expected state from executing `memcmp` with different memory regions. By adding function coverage as a metric in our IOVecs, we could recognize that much of the functionality is not being exercised, and we could pursue other IOVec generating strategies.

Another possibility is for the analyst to provide IOVecs that distinguish the functions in the equivalence class. Generally, this can be difficult to do, but in cases like this, such a task is feasible even for non-experts. The reason `memchr` functions (which returns a pointer) are grouped with `memcmp` functions (which do not return a pointer), is because our fuzzer never generated a memory region containing the value specified in the second argument to `memchr`. Thus `memchr` always returns 0 (which Tinbergen recognizes as a non-pointer) for every accepted input program state. Creating just one IOVec which has a memory region containing the value of the second argument is trivial, but would be enough to distinguish `memchr` from the other functions in this equivalence class. It is similarly trivial to create IOVecs that distinguish between `memcmp` and `strncmp`.

The IO-related functions highlight the impact of process state dependence. All of these functions take a `FILE*` as the first argument, which requires opening a file descriptor beforehand. Tinbergen does not open any file descriptor before fuzzing, and thus the first argument will *never* be semantically correct. Luckily, it is simply an engineering effort to incorporate setting up an appropriate process state before fuzzing, and we also believe that improvement would significantly increase accuracy.

IOVec Extensions. An interesting extension to the information collected in an IOVec is which system calls are invoked while a FUT is executing. System calls are interesting because, as mentioned in § 3, we currently do not capture kernel state in an IOVec; however, system calls serve as a proxy for this state. Additionally, because system calls provide services that cannot be provided by user-space code and cannot be optimized out, semantically equivalent functions *must* invoke the same set of system calls. Therefore, by adding system call knowledge into IOVecs, we expect $\overline{N}$ to decrease, with N remaining unchanged. We would also expect our accuracy to increase, because when confirming a function at the leaf, the probability of the FUT matching the system calls of the equivalence class at the leaf is generally lower.

To test this hypothesis, we added support for tracking which system calls are made during execution, but we did not track the order or number of system calls. Order and number are not tracked, because they do not necessarily translate to semantic equivalence (e.g., calling `read(fd, 1)` 4 times could be the same as calling `read(fd, 4)` once). We then extended our definition of matching program states (see § 3.1) to include checking that the exact same system calls are made (and no more or fewer), and generated a new `gcc O2` decision tree using the same binary in the original evaluation. The total amount of developer effort to add system call support was 73 LoCs. As expected, N remained the same, $\overline{N}$ was decreased by 13%, and accuracy increased by 3.1%. The decrease in $\overline{N}$ is significant, and the gains come from being able to differentiate more functions in the largest of equivalence classes. However, the increase in accuracy is within a standard deviation of the original evaluation, and we thus cannot exclude the possibility of measurement noise.

7 DISCUSSION

Here we provide discussion on unknown functions and the limitations of Software Ethology, as well as some possible future work.

Limitations. We have identified a few sets of functions that Software Ethology is unlikely to classify or identify correctly. These functions are highly dependent upon the system environment while generating IOVecs, as well as during the identification phase. Functions like `getcwd` or `getpid`, which return the current working directory and the process ID of the analysis respectively, depend on the filesystem, current user, and kernel state. As these factors differ between runs or are non-deterministic, our fundamental assumption — semantically similar functions change their program state in similar ways given a specific input program state — breaks down. To address this limitation, Tinbergen could model the system state in addition to the process state.

Another set of functions Software Ethology struggles with depend on an initial seed being set beforehand. Examples of these functions include `rand` and `time`. As we execute functions without any knowledge about their behavior, we cannot provide the seed beforehand as it is difficult to distinguish a seed value from other global variables. Even if we are able to determine a location of the seed, knowledge of proper API usage (e.g., calling `srand` before `rand`) is needed to correctly use these functions. Discerning correct API usage is an active research area [1], and improvements in this area will directly translate to improvements in Tinbergen.

Unknown Functions. If a function is encountered that accepts no known *DCIS*, Software Ethology will mark this function as

unknown. When a function is marked as unknown, it can mean one of two things depending on the number of accepted IOVecs. If the unknown function *never* accepts an IOVec, then it implements wholly unknown functionality, and should be a main focus for analysts. Otherwise, if the function accepts some IOVecs, then it shares some functionality with the functions whose *DCIS* includes the accepted IOVecs. The utility analysts might gain from this information varies with the number of IOVecs accepted. Many IOVecs rejected with a few IOVec acceptances is likely a common failure mode present in many functions, e.g., returning −1 on invalid input. If many IOVecs in a *DCIS* are accepted, then the unknown function is likely similar to the corresponding function, indicating, e.g., a different version.

Future Work. Tinbergen generates a function's *DCIS* through the use of techniques employed by mutational fuzzers [6, 22, 24, 35, 62]. Another key component employed by mutational fuzzers (but currently unused by Tinbergen) is the use of code coverage to guide fuzzing. By incorporating code coverage, it is possible to minimize the *DCIS* that provides 100% edge or code coverage of a function, designated as $DCIS_\Omega$. Later, if that function is identified in a new binary, then any deviation in code coverage when given $DCIS_\Omega$ would indicate the presence or lack of functionality in the FUT. This could be helpful in exploit generation, or code version identification [5]. Recent work in binary instrumentation [9, 58] make dynamically recording code coverage feasible.

There are several other avenues of future work to extend Software Ethology. The natural next step is to add support for arguments passed on the stack and global variables. Stack-based arguments can be supported by, for example, writing a data to an valid memory location, and adjusting the stack and base pointers appropriately pre-execution. More care is needed for handling global variables, but it might suffice to simply record the `bss` and `data` segments before and after execution.

We would also like to pursue support for C++ binaries, specifically determining what classes are present in a binary. Currently, the `this` pointer — an ever-present part of the program state when class methods are executed — presents a unique challenge, as the underlying memory object is expected to have a vtable and members in particular locations. Additionally, objects are always expected to have their constructor called before any method, and can have other objects as class members.

A challenging aspect of reverse engineering is the detection of cryptographic functions in a binary. They are difficult to identify, because they are often implemented using architecture-specific assembly for optimization purposes, make extensive use of randomness, and rely heavily on correct state and input. These are situations for which Software Ethology is particularly well-suited, and it would be worthwhile to investigate how far we can advance automated analysis on these most difficult class of functions. IOVecs, as an extension of captured state, could record the random values returned by RNGs. In the coalescing and identification phases, calls to RNGs could be intercepted, and the recorded random value could be returned instead of a "true" random value.

8 RELATED WORK

Similarity analysis is an active area of research [7, 13, 17, 20, 33, 42]. Jiang et al . [28] first proposed using randomized testing in function similarity analysis, drawing inspiration from polynomial identity testing. Their system requires source code to identify syntactically different yet functionally similar code fragments, and is able to scale to large code bases such as the Linux kernel.

`BLEX` [16], the closest research to our proposal, extracts features of binary functions by guaranteeing that every instruction is executed. Similarity is determined by computing an normalized weighted sum of the Jaccard indices between two function's feature vectors. They also implemented a search engine with their system similar to Tinbergen. Wang, et al. [59] perform code similarity analysis using a system called `IMF-SIM`. `IMF-SIM` uses an in-memory fuzzer to measure the same metrics as `BLEX`, instead of forcing execution to start at unexecuted instructions. `IMF-SIM` computes a longest common subsequence of recorded features, and makes a similarity decision based on a consensus of randomly generated trees of feature vectors. As stated in our evaluation, these works still struggle with differing compilation environments, while Tinbergen has consistently high accuracy irrespective of compilation environment. Both works focus on measuring code properties, which change with different compilation environments. Software Ethology, in contrast, uses IOVecs, which are independent of code.

Su et al. [57] use dynamic analysis to compute a dependency graph between instructions. Code similarity is determined by computing an isomorphism between subgraphs. Their system, DyCLINK, targets Java applications with no support for native applications or libraries, and thus, we cannot compare Tinbergen against DyCLINK. DyCLINK suffers from the same limitations as `BinDiff`, namely that computing graph isomorphisms is NP-Complete. While heuristics can provide respite from state explosion in some cases, a general polynomial time algorithm remains elusive or impossible.

Due to the diverse toolchains and architectures used and its closed-source nature, binary analysis is particularly well suited to firmware. David et al. [14], created a static analysis tool to find CVEs in firmwares, and discovered hundreds of vulnerabilities. Feng et al. [18], took inspiration from picture and video search to create a system that finds bugs in Internet of Things devices by converting CFGs into numerical vectors to analyze.

Recently, neural networks have been used in binary analysis. In a paper published on arXiv, Zuo et al. [63], trained a neural network to determine cross-architecture semantics of basic blocks. Liu et al. [37], employs a deep neural network to extract features from functions and the binary call graph to compute. These features are then used to create a distance metric for determining binary similarity. Xu et al. [61] use a neural network to compute the embedding of a function's CFG to accelerate similarity computation. These approaches show promise in improving computer security by utilizing research from other fields in computer science.

9 CONCLUSION

We introduce Software Ethology, a semantic function identification framework that is compilation environment agnostic. Instead of measuring code properties, Software Ethology abstracts functions into sets of input and output program states, information

guaranteed to be stable across compilation environments. Our proof-of-concept, Tinbergen, has a high accuracy when identifying functions in binaries generated from various configurations, while being easy to use in practice. Tinbergen is .805 accurate across compilation environments, and is 50% faster than state-of-the-art at classifying binaries. We will release Tinbergen as open source upon acceptance.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.

[2] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 177–189.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. (2019). https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[4] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, behavior-based malware clustering.. In *NDSS*, Vol. 9. Citeseer, 8–11.

[5] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*. IEEE Computer Society, Washington, DC, USA, 143–157. https://doi.org/10.1109/SP.2008.17

[6] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 725–741.

[7] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-architecture cross-OS Binary Search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 678–689. https://doi.org/10.1145/2950290.2950350

[8] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[9] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries.. In *NDSS*. Citeseer.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3133956.3134069

[11] Intel Corporation. 2018. Pin - A Dynamic Binary Instrumentation Tool. (2018). https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[12] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 266–280. https://doi.org/10.1145/2908080.2908126

[13] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 79–94.

[14] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 392–404.

[15] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1455770.1455779

[16] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 303–317. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

[17] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. https://doi.org/10.14722/ndss.2016.23185

[18] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 480–491. https://doi.org/10.1145/2976749.2978370

[19] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable Detection of Semantic Clones. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 321–330. https://doi.org/10.1145/1368088.1368132

[20] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-platform Binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 896–899. https://doi.org/10.1145/3238147.3240480

[21] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

[22] Google. 2018. honggfuzz. (2018). https://github.com/google/honggfuzz

[23] Google. 2018. OSS-Fuzz. (2018). https://github.com/google/oss-fuzz

[24] Gustavo Grieco, Martí n Ceresa, and Pablo Buiras. 2016. QuickFuzz: An Automatic Random Fuzzer for Common File Formats. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 13–20. https://doi.org/10.1145/2976002.2976017

[25] SA HexRays. 2019. Interactive Disassembler. (2019). https://www.hex-rays.com/index.shtml

[26] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX , Washington, D.C., 81–96. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/jang

[27] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 96–105. https://doi.org/10.1109/ICSE.2007.30

[28] Lingxiao Jiang and Zhendong Su. 2009. Automatic Mining of Functionally Equivalent Code Fragments via Random Testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/1572272.1572283

[29] Ryan Johnson and Angelos Stavrou. 2013. Forced-path execution for android applications on x86 platforms. In *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*. IEEE, 188–197.

[30] Dave Jones. 2018. Trinity : A Linux system call fuzzer. (2018). http://codemonkey.org.uk/projects/trinity/

[31] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.* 28, 7 (July 2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[32] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM '07)*. ACM, New York, NY, USA, 46–53. https://doi.org/10.1145/1314389.1314399

[33] Ulf Karg é n and Nahid Shahmehri. 2017. Towards Robust Instruction-level Trace Alignment of Binary Code. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 342–352. http://dl.acm.org/citation.cfm?id=3155562.3155608

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[35] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

[36] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code.

[37] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. α Diff: Cross-version Binary Code Similarity Detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 667–678. https://doi.org/10.1145/3238147.3238199

[38] LLVM. 2019. libFuzzer. (2019). https://llvm.org/docs/LibFuzzer.html

[39] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Trans. Softw. Eng.* 43, 12 (Dec. 2017), 1157–1177. https://doi.org/10.1109/TSE.2017.2655046

[40] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 24–35.

[41] Reed Milewicz, Rajesh Vanka, James Tuck, Daniel Quinlan, and Peter Pirkelbauer. 2015. Runtime checking c programs. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2107–2114.
[42] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 253–270. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ming
[43] musl libc. 2019. musl libc. (2019). https://www.musl-libc.org/
[44] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.
[45] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-force: force-executing binary programs for security applications. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 829–844.
[46] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.
[47] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 709–724.
[48] Rui Qiao and R Sekar. 2017. Function interface analysis: A principled approach for function recognition in COTS binaries. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 201–212.
[49] radare. 2019. radare. (2019). https://www.radare.org/r/
[50] Thomas Rid and Ben Buchanan. 2015. Attributing cyber attacks. *Journal of Strategic Studies* 38, 1-2 (2015), 4–37.
[51] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. 2006. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 289–300.
[52] Andreas S æ bj ø rnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting Code Clones in Binary Executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1572272.1572287
[53] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo
[54] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.
[55] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society, 1–15.
[56] Richard M. Stallman and GCC DeveloperCommunity. 2009. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA.
[57] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code Relatives: Detecting Similarly Behaving Software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 702–714. https://doi.org/10.1145/2950290.2950321
[58] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making reassembly great again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSSfi17)*.
[59] Shuai Wang and Dinghao Wu. 2017. In-memory Fuzzing for Binary Code Similarity Analysis. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 319–330. http://dl.acm.org/citation.cfm?id=3155562.3155606
[60] W. Xu, H. Moon, S. Kashyap, P. Tseng, and T. Kim. 2017. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, Vol. 00. 577–593. https://doi.org/10.1109/SP.2019.00035
[61] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 363–376.
[62] Michal Zalewski. 2015. American Fuzzy Lop. (2015). http://lcamtuf.coredump.cx/afl/
[63] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).
[64] zynamics. 2019. BinDiff. (2019). https://www.zynamics.com/bindiff.html

A ADDITIONAL INFORMATION

```
xcharalloc
__munmap
close
dup2
c_isalpha
```

Figure 11: A selection of commonly mislabeled functions.

```
void fuzz_strategy(uint8_t *buffer) {
    int choice = rand()
    if (choice == 0)
        flip_bit_at_random_offset(buffer);
    else if (choice == 1)
        set_interesting_byte_at_random_offset(buffer);
    else if (choice == 2)
        set_interesting_word_at_random_offset(buffer);
    else if (choice == 3)
        set_interesting_dword_at_random_offset(buffer);
    else if (choice == 4)
        inc_random_byte_at_random_offset(buffer);
    else if (choice == 5)
        inc_random_word_at_random_offset(buffer);
    else if (choice == 6)
        inc_random_dword_at_random_offset(buffer);
    else if (choice == 7)
        set_random_byte_at_random_offset(buffer);
    else if (choice == 8)
        set_random_word_at_random_offset(buffer);
    else if (choice == 9)
        set_random_dword_at_random_offset(buffer);
}

```

Listing 7: How Tinbergen fuzzes a register or allocated memory region.

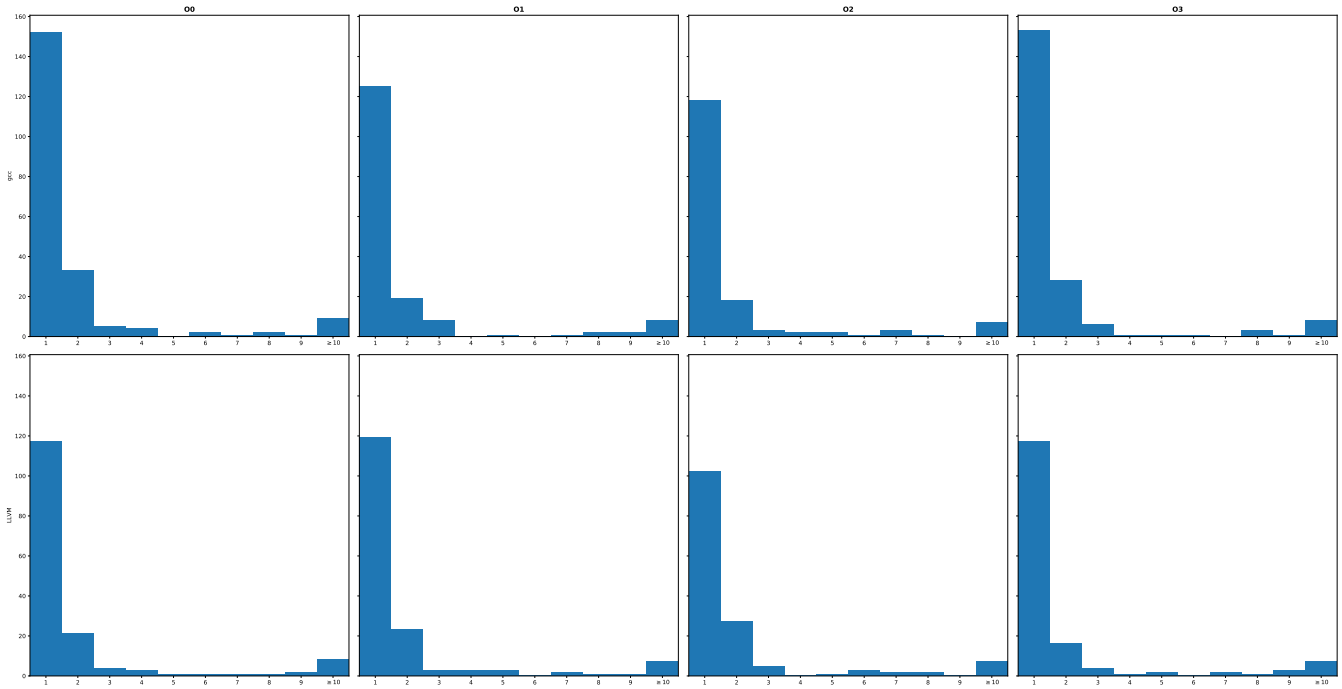

Figure 12: The distribution of equivalence class sizes for all generated decision trees in the `coreutils-8.30` evaluation.