RevEngE is a dish served cold: Debug-Oriented Malware Decompilation and Reassembly

Marcus Botacin
mfbotacin@inf.ufpr.br
Federal University of Paraná (UFPR-Brazil)

Lucas Galante
galante@lasca.ic.unicamp.br
University of Campinas (UNICAMP-Brazil)

Paulo de Geus
paulo@lasca.ic.unicamp.br
University of Campinas (UNICAMP-Brazil)

André Grégio
gregio@inf.ufpr.br
Federal University of Paraná (UFPR-Brazil)

ABSTRACT

Malware analysis is a key process for knowledge gain on infections and cybersecurity overall improvement. Analysis tools have been evolving from complete static analyzers to partial code decompilers. Malware decompilation allows for code inspection at higher abstraction levels, facilitating incident response procedures. However, the decompilation procedure has many challenges, such as opaque constructions, irreversible mappings, semantic gap bridging, among others. In this paper, we propose a new approach that leverages the human analyst expertise to overcome decompilation challenges. We name this approach "DoD—debug-oriented decompilation", in which the analyst is able to reverse engineer the malware sample on his own and to instruct the decompiler to translate selected code portions (e.g., decision branches, fingerprinting functions, payloads etc.) into high level code. With DoD, the analyst might group all decompiled pieces into new code to be analyzed by other tool, or to develop a novel malware sample from previous pieces of code and thus exercise a Proof-of-Concept (PoC). To validate our approach, we propose RevEngE, the Reverse Engineering Engine for malware decompilation and reassembly, a set of GDB extensions that intercept and introspect into executed functions to build an Intermediate Representation (IR) in real-time, enabling any-time decompilation. We evaluate RevEngE with x86 ELF binaries collected from VirusShare, and show that a new malware sample created from the decompilation of independent functions of five known malware samples is considered "clean" by all VirusTotal's AVs.

ACM Reference Format:
Marcus Botacin, Lucas Galante, Paulo de Geus, and André Grégio. 2019. RevEngE is a dish served cold: Debug-Oriented Malware Decompilation and Reassembly. In *Proceedings of ACM ROOTS (ROOTS 19).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Malware analysis is a key task for gathering information on infections, since it enables security countermeasures such as the development of vaccines [37], incident response procedures [51], etc. Malware analysis solutions have been evolving from dynamic tracers [3, 27, 60] to complete code decompilers [24, 42, 49], which may allow the discovery of execution behaviors or potentially more detailed capabilities in the source-code, respectively.

Binary decompilation is already challenging for "ordinary" code. Malware decompilation can be even more challenging, since (i) instruction disassembly is difficult to accomplish if data and code are mixed [47], or the developer used opaque constants for code obfuscation [34]; (ii) instructions might be context-dependent [29] (e.g., CPU-dependent) and malware often rely on these instructions for fingerprinting procedures [6]; (iii) handling actual COTS binaries is hard because the x86 ISA is very large and presents broad corner conditions that limit decompilation inferences [17]; (iv) malware can use multiple calling conventions in the same binary, which complicates the identification of function prototypes [33]; (v) evaluating the decompilation results may become extremely expensive due to the amount of dead code that can be embedded in malware samples [59]. To overcome the aforementioned challenges on decompiling malware, we introduce a debug-centric approach, which leverages the analysts knowledge to support decompilation decisions. In the debug-centric modus operandi, the analyst starts by debugging a malware sample and asking for the decompilation of a given code region (e.g., code function). Each code region can be decompiled more than once, according to the analyst's provided parameters and the execution paths she choose to follow. Therefore, the decompilation does not reflect the binary content, but the investigation steps conducted by the malware samples' analyst. Thus, decompiled code pieces can be used to generate new malware PoCs for more detailed security analysis, or even offensive purposes, such as malware re-engineering.

We also introduce RevEngE[1]—the Reverse Engineering Engine for malware decompilation and reassembly—as a tool to evaluate our debug-centric approach. RevEngE consists of GDB extensions that intercept and introspect-into executed functions to build an Intermediate Representation (IR) of the analyzed sample in real-time, which allows that decompilation occurs at any time of the execution. Overall, RevEngE addresses the listed decompilation challenges by relying on: (i) dynamic inspection, for sorting out data from instructions; (ii) GDB, to avoid the reimplementation of x86 instruction handling support; (iii) the analyst knowledge, for the definition of decompiled code slices; and (iv) the evaluation of decompilation outcome in terms of malware reassembly capabilities instead of recovered code. We implemented RevEngE in Python and exploited Object-Oriented-Programming (OOP) capabilities to handle x86 instruction heterogeneity via polymorphic constructions and operators overloading. We also implemented a network

ROOTS 19, 2019, Vienna
2019.

[1] No relation to the https://rev.ng disassembler

module that allows RevEngE to introspect into linked-libraries and discover function prototypes based on similar constructions found on the Internet. We evaluated RevEngE with x86 ELF malware samples collected from VirusShare [56] and a newly crafted malware sample based on independent functions taken from five known malware. Our developed malware was not detected by any AV on VirusTotal [57] at the time of this writing, yet the malware samples it was based on were detected by multiple AVs. The obtained results show that our proposal of debug-oriented decompilation is feasible, as well as useful as a dual tool either for malware analysts and malware developers.

Our contributions in this paper are:

- we discuss the challenges and limits of malware decompilation procedures. Whereas previous work have already individually pinpointed these challenged, they are here grouped and presented according their occurrence during the steps of development of an actual malware decompiler;
- we introduce the debug-oriented approach to assist malware decompilation;
- we propose RevEngE, a PoC to perform debug-oriented decompilation, and detail its design and implementation; **RevEngE code will be open-sourced after publication acceptance.**
- We showcase examples of RevEngE usage to decompile actual malware samples for defensive and offensive purposes.

The remainder of the paper is organized as follows: in Section 2, we revisit malware analysis solutions to better position our contributions; in Section 3, we discuss malware decompilation challenges; in Section 4, we present RevEngE's design and implementation; in Section 5, we evaluate RevEngE by decompiling actual malware; in Section 6, we discuss RevEngE's advantages and limitations; finally, we draw our conclusions in Section 7.

2 RELATED WORK: MALWARE ANALYZERS

We here shown an overview on how malware analysis tools have been evolving from sandboxes to decompilers to better position RevEngE among previous work, and how it relates to recent advances in decompiling.

First Generation Analyzers are sandboxed solutions (e.g., Anubis [3, 27], CWSandbox [60]) to spot malware behavior by running the sample and extracting indicators of compromise (IoCs) that might help further incident response procedures. These analyzers neither build high level representations of malware samples, nor address multiple execution paths.

Second Generation Analyzers represent the information extracted from malware sample's execution using an Intermediary Representation (IR) [1, 55] (e.g., the VINE component of Bitblaze [50], BAP framework [7]). Thus, they enable additional tracing capabilities through the observation of multiple execution paths or constructions.

Third Generation Analyzers leverage IR to extend their analysis capabilities from only tracing to additional inspection resources, such as Angr.io [48], which builds a program tree to exploit bugs; HexRays [24] and Snowman [49], which implement full decompilers; ERESI [16], which proposes an IR-based debugger; RADARE [42], which mixes debugging, tracing and decompilation capabilities. Our solution is a third generation analyzer.

Recent Decompilation Advances. Third-generation analyzers are powered by recent developments in the decompilation field. More specifically, dynamic inspection approaches allow solutions to follow multiple execution paths, thus overcoming decompilation challenges such as reconstruction of data structures [12], data types [54], and loop information [45]. In this work, we adopt a dynamic decompilation approach via debugger instrumentation.

Previous work suggested that interactive debugging procedures could be used to assist decompilation by increasing code coverage [19], or that trace-oriented programming could help in understanding binary behavior [62]. We extended these works for the specific case of malware decompilation. Though decompilation have already been applied for algorithms identification within unknown binaries [36] and for malware analysis [61], we go one step further and propose to reassemble malware decompiled functions and algorithms into new pieces of code.

3 BACKGROUND: COMPILERS & DECOMPILERS

In this section, we show how compilers and decompilers operate, and discuss challenges of malware decompilation (some of them tackled by RevEngE).

3.1 Similarities & Differences

A **compiler** is a tool that transforms high-level code into low-level representation of it. In this work's context, it takes a code written on a high-abstraction programming language (e.g., in C) as input and generates a machine understandable code. A typical compilation procedure is divided into the following steps: *parsing*, in which an input file has its content loaded into memory in a convenient representation; *pre-processing*, which expands macros and constants, and propagate them along the code (e.g., constants like `#define N 10` are consolidated on expressions, such as `for(i=0;i<N;i++)`); *code generation*, which performs high-level code traversal so the compiler may emit lower-representations code (assembly) according to the identified control-flow structures; *assembling*, in which the produced code is translated to actual machine code; *linking*, which resolves external function calls/symbols on binary relocated sections.

A **decompiler** is a solution that turns low-level code into a representation in high-level. In this work's context, it transforms machine code into a human-readable representation, thus being sometimes referred as inverse compiler [11]. As compilation, the decompilation procedure can be divided into small steps: Hollander [25] names decompilation steps as `init, scan, parse, construct, generate`, whereas the HexRays decompiler [24] adopts `disassembly, lift, data type recovery, code generation`. Other steps are defined by Serrano [47]. Despite different naming schemes, decompilation steps are very similar: it starts with the `disassembling` of a given binary or the `parsing` of disassembly data taken as input; the `lift` phase consists of raising assembly code to an intermediate representation; `data type recovery` adds meaning to data values; if `lift` and `data type recovery` are combined, the result is the `construct` step. Finally, there is a `code generation` routine that

produces high-level code, instead of machine code produced by compilers.

Backend vs Frontend. The internals of compilers and decompilers are frequently divided into `frontend` and `backend`. A compiler frontend is machine-independent and responsible for handling high-level constructs, while its backend is machine-dependent and responsible for code-generation. As inverted compilers, decompilers' frontend and backend are reversed, i.e., its frontend handles machine data whereas the backend is machine-independent and handles high level constructs.

3.2 Decompilation Challenges

Disassembly. It is a key step for the decompilation procedure, since code instructions define the behavior of a program. Most decompilers adopt static disassembly approaches, which may be problematic when handling malware samples [47]—they often employ anti-disassembly tricks to bypass static analysis procedures, such as opaque constants [34]. Drawbacks of static disassembly procedures include sorting out instructions from data [12, 26], separating pointer addresses from constant and offsets [55], or the presence of context-dependent instructions (e.g.,`cpuid`) in the assembly code handling [29]. All issues are often tied to malware samples, either in the code construction or for fingerprinting [6]. The challenges are made even bigger when overlapping instructions are observed during the disassembly phase [5], which can be implemented, for instance, for self modifying code malware samples.

A possible solution to these issues is to rely on dynamic execution traces as data sources, which solves data dependencies and data types in runtime [54]. On the one hand, dynamic approaches naturally explicit pointers and function returns [55], thus solving most static analysis issues. On the other hand, dynamic malware inspection approaches suffer from the same limitations of typical malware sandboxes (e.g., evasion due to the lack of transparency [14]), which requires specialized debuggers to be effective [63]. For RevEngE, we adopted dynamic disassembly and implemented debug extensions to armor it against evasive malware. Another issue related to dynamic approaches is ensuring code coverage, since malware samples may require user interaction to take the proper paths, i.e., those that result in the malicious actions. While previous dynamic approaches addressed code coverage by taint tracking user inputs [19, 36], RevEngE relies on analyst interaction with the analyzed code. Dynamic tracing solutions record executed instructions instead of the code structure, making that the `K` instructions within a given `K`-long loop be presented `N` times. These `K`-instruction blocks should be identified and then re-rolled to reconstruct the loop structure, which may be a problem [62]. Existing re-rolling algorithms are used either for loop recovering [52] as for other constructions, such as `break` and `continue` [15]. However, RevEngE adopts a distinct solution that represents code within a loop through Single Statement Assignments (SSA) [55], allowing for the representation of the analyst's interaction with each loop iteration individually. The major issue about loop unrolling and nested function calls serialization is that the trace size might become large until its computation become unfeasible. It is even more concerning if we consider that malware samples often add useless instructions to accomplish stalling behavior [30]. State-of-the-art reverse engineering techniques [9] rely on program slicing to overcome the trace size increasing, since each slice is a semantically meaningful portion of the program that captures a subset of its computation [58]. The challenge here is to determine the length of each slice so as to better capture the program behavior [4] even in face of the presence of non-standard, rarely seen constructions. While previous work suggested using dynamic tracing solutions to automate the slicing size definition [65], RevEngE relies on the analyst's expertise for such task.

Instruction Lifting. Working with x86-like instruction set may be hard due to its CISC architecture, because it also allows memory as operands, in addition to registers. To make instruction handling easier, most decompilers perform some kind of lifting. For instance, Zynamics chose to move instructions to a RISC-based representation, whereas HexRays opted for Intermediate Language (IR) [47]. Although recent solutions insist in directly translate assembly to C code [43], the `asm2c` tool claims that it is unfeasible [7]. Therefore, the design of an appropriated IR is essential for correctly handling binary data, such as the currently widely adopted SSA representation (also used in RevEngE). The large size of the x86 instruction set [2] makes instruction lifting challenging, since it would require a great amount of development effort to parse and support all instruction possibilities. Thus, most decompilers chose to handle only a small subset of instructions and behaviors: REIL [17] supports only 17 of more than 600 available instructions in the x86 architecture; BAP [7] does not handle floating point instructions. These choices may not be reasonable when we need to address malicious binaries, which can use such type of instructions for fingerprinting. RevEngE relies on GDB support to handle the entire x86 instruction set, even when an specific instruction is not implemented in our IR. Heterogeneity is also an issue if we consider the multiple calling conventions that might be used within the same binary or bytecode [33]. While the state-of-the-art for calling convention static determination is based on neural networks [10], RevEngE accomplishes dynamic execution through the underlying GDB support.

Data Type Reconstruction. During decompilation, the instruction-level data need to be translated to a high semantic level, thus allowing the reconstruction of variable types, arrays, and pointers. Data type identification may require solving complex satisfiability expressions [44]. Similar issues can be found in function parameters [12] and returns [64] identification. Addressing them in practice involve the use of heuristics. In RevEngE, we overcome such issues by relying on the underlying GDB platform. Besides internal function calls, decompilers must reconstruct data type from external function calls (e.g., dynamically linked libraries) to handle complex code constructions. The lack of symbols and additional information makes this task hard to accomplish. To do so, current approaches make use of automated solutions that parse known libraries in the search of required information [35]. RevEngE bridges the semantic gap by querying the Internet for function prototypes when it identifies a call for a linked library function.

Code Generation. The last step of the decompilation procedure, code generation is responsible for representing data in a human-readable way. However, its correctness is largely tied to the results from previous steps. In comparison to other steps, there are few

works on improving decompiled code generation [15]. In RevEngE, we exploit Python's polymorphism capabilities to implement a new code-generation strategy that allows the decompilation to occur at any time during program's debugging.

Decompiler Evaluation. Reconstruction time and quality of the recovered code are some of the metrics used to evaluate a decompiler [46]. However, the amount of converted code is not a good metric, since "the core problem of decompilation resides on the remaining parts" [59] and malware may insert dead code to pollute the produced code In this work, instead of considering the retrieved code after decompilation, we evaluate the ability of the decompiler in malware reassembling.

4 REVENGE: DESIGN & IMPLEMENTATION

We here introduce the development of RevEngE, a **PoC** realizing the debug-oriented decompilation approach. RevEngE's goal is not to implement all decompiler capabilities but the minimal requirements to streamline the debug-oriented decompilation procedure. RevEngE's ultimate goal is to produce a reassembled malware piece that might be used in practice for multiple purposes, such as forensic procedures, testing antivirus engines against malware variants, or even to find bugs in malware samples [8].

4.1 System Model

RevEngE is a PoC tool developed to validate our proposed approach of debug-oriented decompilation. It consists of two components: (i) the native GDB as the decompiler frontend to process userland assembly code; and (ii) a Python-based GDB plugin as the decompiler backend to produce C code. Whereas ELF malware might be implemented in multiple languages, we focus on C-like binaries as they are currently the most prevalent ELF malware types [13],

In RevEngE operation model, the analyst starts to debug a malware sample in GDB in the search of interesting functions and code constructions. The analyst then asks RevEngE to decompile function excerpts, which are added to a list of decompiled functions. Each function can be decompiled more than once (according to the provided parameters and followed execution paths), thus generating distinct decompiler outcomes. As in any debugging procedure, the analyst can define the execution paths to be followed by calling and stepping functions. Hence, RevEngE decompilation does not reflect the binary content, but the analyst investigation flow. Analysts may even decompile a code before the function ends, generating a piece of code completely distinct of the original one. After all functions are decompiled, they can be grouped into a final high-level source-code file, and reorganized to build a different program. Rebuilt programs may be used to both defensive and offensive purposes, such as to aid in detection heuristics development through the analysis of unknown malware or to create a malware variant capable of evading security solutions.

The goal of RevEngE is not to be the definitive malware decompiler, but to pinpoint ways to overcome the following decompilation challenges: (1) static disassembly limits via debug-oriented, dynamic instruction handling; (2) lack of stimulation through analyst-malware interaction; (3) loop unrolling problem, by using SSA definitions; (4) code slice size definition outsourced to analysts; (5) parsing of all x86 instructions with GDB even when RevEngE does not have an IR class for such instruction, (does not break malware execution); (6) real-time IR generation, i.e., decompilation may occur at any debug time; (7) multiple calling convention support via outsourced GDB identification (allows inspection using multiple calling conventions in the same binary); (8) dynamic code retrieval to recover only code actually executed, mitigating issues related to dead-code insertion in malware; (9) introspection in linked libraries function calls through automated semantic gap bridging that retrieves function prototype definitions from the Internet; and (10) exploiting Python OOP paradigm with polymorphic methods to overwrite code generation and emit high-level C code according to each x86 instruction behavior.

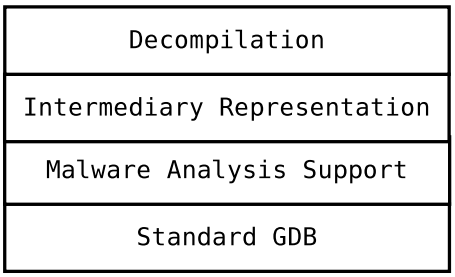

Figure 1: RevEngE Architecture. GDB provides the basic debugging capabilities and was armored to handle malware anti-analysis techniques. RevEngE decompiler is developed on top of the armored GDB.

To simplify RevEngE implementation, we limited the malware operation context by assuming that the code under decompilation: (i) performs computations using only integer values (no floating-point operations supported); (ii) can reference integer vectors, but each vector bucket will be decompiled to a distinct variable; (iii) does not perform signed/unsigned conversions and all numbers are represented as 2's complement; (iv) can reference static or dynamic chars, but the decompiler only generates static char declarations, without dynamic allocation support; (v) can use loop constructions, but these will be unrolled during decompilation; (vi) does not rely on predicated instructions, since handling speculative execution is hard. Unlike other approaches, we do not prevent the malware from performing such actions, but limit the generation of decompiled code encompassing such constructions.

4.2 System Architecture

RevEngE architecture has four layers (Figure 1). The first layer is GDB itself. By relying on an existing debugging solution, RevEngE does not need to re-implement inspection features, such as breakpoints, disassembly and other basic tasks required for binary analysis.

The second layer is composed by a series of malware analysis and inspection extensions, which are require to armor GDB against "tricky" malware constructions as the GDB was originally designed to handle only goodware samples. RevEngE implements three analysis extensions: (i) Automatic entry point identification, to inspect stripped binaries; (ii) Control flow modification support, to skip anti-analysis checks; and (iii) Skipping ptrace calls, to avoid anti-debug techniques. Whereas this is not an exhaustive list of anti-analysis

tricks, these analysis features are enough to handle most samples present in our dataset, thus enabling RevEngE evaluation.

The third layer is composed by the IR implementation, as low level assembly instructions must be lifted to a high-level representation. As our approach is runtime-based, this step is also responsible for keeping debugging state, as the high-level representation may change as the debugging procedure steps.

Finally, the upper layer is responsible for C code emission. Unlike other layers, which are affected by each instruction step, RevEngE only decompiles code on-demand.

4.3 Implementation

We here describe how RevEngE implements its functions on each one of the previously presented layers.

GDB Instrumentation. GDB did not provide native support for extensions. To overcome this limitation, GDB developers integrated a Python interpreter within it, thus allowing scripts to be run in GDB's context. RevEngE leverages this feature to implement its analysis resources.

This strategy was already deployed by other GDB-based solutions, such as PEDA [38], Pwndbg [40] and GEF [21]. In common, they are all intended to be used as exploitation frameworks. As far as we know, we are the first to present a GDB-based decompiler.

RevEngE collects data by parsing GDB outputs, wrapping GDB commands in RevEngE commands.

Malware Analysis Support. RevEngE implements some debugging extensions to assist malware analysis procedures. The extensions are not directly related to the decompilation procedure but they help analysts while selecting the code regions which will be decompiled. Developing such support is required as GDB is not designed to natively handle malware, thus not offering all resources required by malware analysts.

Automatic Entry Point Identification. RevEngE's key idea is to allow an analyst to mark the code regions that he/she is debugging for decompilation. However, if one runs a binary without setting breakpoints, the execution will finish without stopping and no code region would be marked and decompiled. To handle this case, RevEngE automatically sets a breakpoint to the `main` function, the entry point of standard binaries. Nevertheless, malware binaries are often stripped, thus not exporting a `main` symbol. To still allow setting an entry point breakpoint even in stripped binaries, RevEngE implements an automatic entry point identification procedure, as follows: (i) RevEngE first retrieves the `libc` address from the ELF header; (ii) sets a breakpoint on the identified `libc` address; (iii) retrieves the first `libc` function call argument, which is the program entry point address, as shown in Code 1; and (iv) sets a breakpoint on the identified entry point address.

```
1 __libc_start_main (main=<value optimized out>, argc=<
      value optimized out>, ubp_av=<value optimized out
      >,
2     init=<value optimized out>, fini=<value optimized
          out>, rtld_fini=<value optimized out>,
3     stack_end=0x7fffffffdc38) at libc-start.c:258
```

Code Snippet 1: Libc Entry Point. First argument points to application entry point.

RevEngE relies in this procedure to automatically stop at the entry point when it identifies that the analyzed binary is stripped. This approach is supported by the fact that we our model assumes that only standard C binaries will be handled, thus the `libc` library will always be present.

Invert Branch Direction. Malware samples often rely on complex code constructions to mask their decision paths. Therefore, a typical analyst task is to follow paths distinct than the ones pointed by samples' native branches. This can be accomplished by inverting directions when a conditional branch is identified.

Branches are taken according the flags set in the processor. When a `cmp` instruction is executed, flags such as `overflow` and/or `zero` are set. Therefore, a way of inverting branch direction is to invert the flags right after the `cmp` instruction, as implemented by RevEngE. Therefore, each time an invert command is raised, the flags register is changed by RevEngE, as shown in Code 2.

```
1 output = gdb.execute("set␣$eflags|=0x%x" % self.
      flag_map[flag],to_string=True)
```

Code Snippet 2: Invert Branch Direction. Flags register is changed according a map of possible flags for such command.

Skipping Ptrace Checks. GDB's inspection capabilities are supported by the ptrace framework, thus a way of evading GDB inspection is to detect ptrace attachment to the monitored process. It can be done by trying to attach ptrace to itself, as shown in Code 3. If the attachment fails, the sample will realize that some other entity (the debugger, in this case) previously attached to it and may evade analysis.

```
1 if (ptrace(PTRACE\_TRACEME, 0, NULL, 0) == -1)
```

Code Snippet 3: Ptrace self-check. RevEngE skips ptrace self-checks to avoid being detected.

To avoid being deceived by this anti-analysis technique, RevEngE allows analyst to skip ptrace checking. RevEngE implements such skipping capability by relying on the previous branch inversion technique to automatically shepherding the application execution flow towards a path free of ptrace calls.

Instruction Handling and IR. RevEngE relies on an Intermediate Representation (IR) to lift instructions from a lower to a higher abstraction level, which is done by representing each instruction as a class object. As in RevEngE's operational model decompilation can occur at any time, each executed instruction's object is responsible to update the current decompilation context, thus immediately updating variable values, both in memory as well as in registers. To avoid re-implementing the same context controlling routines in each instruction, we explored the Python OOP characteristic to implement polymorphic classes definitions, thus handling all arguments in the same way and only rewriting the code emission method to reflect instruction operation. Figure 2 exemplifies this project decision for the arithmetic functions: `ADD` and `SUB` are handled the same way, via their parent class' methods, but their code emission methods is redefined to emit the proper high-level operators (+ and -, respectively).

RevEngE also benefits from Python OOP capabilities to handle the arguments heterogeneity enabled by the x86 CISC instruction set. RevEngE overloads classes constructors to provide support for

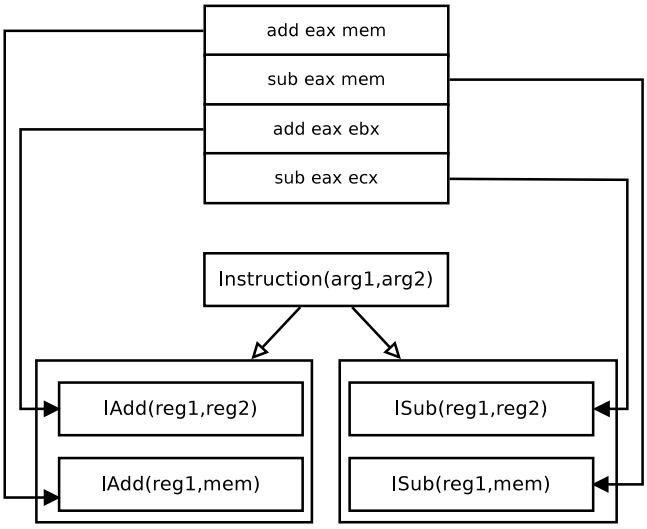

Figure 2: Instruction Representation. RevEngE benefits from Python's polymorphism to model instruction's behaviors and overloads method declarators to support each x86 instruction's possible multiple argument types.

multiple argument types combinations, such as registers (`reg`) and memory (`mem`) operands, as also exemplified in Figure 2.

RevEngE implements a factory design pattern [20] to allow the proper object class selection, as shown in Code 4. Therefore, instead of directly creating a new object, RevEngE asks a instruction factory to return the proper object based in the provided arguments (register, memory, or constants).

```
class IFactory(...):
    def get(self, args):
        newclass = globals()[name](args)
        return newclass
```

Code Snippet 4: Instruction Factory. The Factory design pattern allows instantiating objects from the proper class by exploring Python OOP capabilities.

A challenge for instruction lifting is that a given high level behavior can be implemented by multiple distinct assembly instruction. RevEngE overcomes this challenge by exploiting model's assumptions. For instance, Code 5 shows that all distinct division instructions are handled by the same (`IDiv`) high-level class, since only signed integers are assumed in RevEngE's model.

```
self.classes['div']   = "IDiv"
self.classes['divl']  = "IDiv"
self.classes['idiv']  = "IDiv"
self.classes['idivl'] = "IDiv"
```

Code Snippet 5: Instruction Lifting. RevEngE assumes only signed integer operations to handle all instructions via the same high-level class.

Some code constructions, however, cannot be directly mapped from single instructions to a single class. Conditionals, like ifs, for instance, are the most noticeable example of this type of construction. Code 6 shows that an if is composed by a compare (`cmp`) instruction that sets the comparison flags and a branch instruction (`jle`) that jumps to a given target if the comparison flag is set.

```
0x4004eb cmp -0x8(%rbp),%eax
0x4004ee jle 4004fb <main+0x25>
```

Code Snippet 6: Low level representation of a conditional decision. IF instructions are composed by multiple assembly instructions.

To handle such complex cases, RevEngE implements a type promotion schema. In such, individual instructions are aggregated into a higher level construction class. Code 7 exemplifies the aggregation of the comparison instruction and the resulting flag into a high level comparison class that represents conditional constructions.

```
class HighLevelCompare():
    def __init__ (self,cmp,set):
        self.op1 = cmp.op1
        self.op2 = cmp.op2
        self.op3 = set.op3
```

Code Snippet 7: High level conditional decision representation. Assembly instructions are promoted to a single class that represents a high level conditional structure (e.g., IFs).

RevEngE supports operations over distinct argument types by outsourcing their manipulation to a variable management class, as shown in Code 8. The manager is responsible for storing all variable information, such as name, assigned register and/or memory position, and for keeping context consistent after instruction's execution, by evicting variables from register to the memory and avoiding duplicated entries.

```
self.vars = VariableManager()
self.vars.remove_registers(reg=arg1.get_operand())
self.vars.check_is_pointer(var.get_value())
```

Code Snippet 8: Variable Management. RevEngE does not handle variables directly but via a centralized manager to keep context consistent.

Code 9 shows that the variable manager encapsulates the complexity of handling the multiple storage possibilities by allowing RevEngE to create or retrieve a variable by using both limited and full variable information, since the manager internally updates all tables to keep context coherent, such as: (i) evicting a variable to memory when its associated register is requested by another operation; (ii) loading a variable from memory to register when it is requested by an operation; (iii) promoting constant values to initialized variables; and (iv) creating new variables when a register or memory value is overwritten;

```
self.var = self.vars.new_var(reg="%eax")
self.var = self.vars.new_var(reg=arg1.get_operand(),
     value=val)
self.var = self.vars.new_var(value=arg1.get_value(),
     mem=arg2.get_operand())
```

Code Snippet 9: Variable Manager. Context complexity is encapsulated by the manager, thus releasing RevEngE to focus on decompilation logic.

A key challenge overcome by the variable manager is to disambiguate memory references. As instructions reference variables in a `register+offset` way, distinct instructions might reference the

same address using distinct registers and offsets. These cases must be identified and properly handled to not break the code semantic by creating a new variable according the pointed register. Whereas this is a challenge for static disassembly solutions, RevEngE address this problem by dynamically inspecting the value pointed by the registers and offset, thus handling variables by their memory addresses and not by the pointed register, thus naturally avoiding conflicts. Code 10 illustrates how RevEngE mapped two distinct references to the same variable.

```
(\revenge) ...da main movl $0xF -0x4(%rbp)
NAME: [var0]  VAL: [0xF]  REG: [NONE] MEM: [7
     fffffffdc7c]
(\revenge) ...0e main mov -0x8(%rbp) %eax
NAME: [var0]  VAL: [0xF]  REG: [NONE] MEM: [7
     fffffffdc7c]
```

Code Snippet 10: Memory References Disambiguation. Variables are referenced by their memory addresses instead of pointed registers.

Handling Special cases. Atypical code constructions may appear naturally during program compilation, due to bugs, or being intentionally created by malware authors. In RevEngE's model of anytime decompilation, atypical code constructions may also be originated by premature function exits. RevEngE handles the following atypical cases: (i) identifying direct references to argc, argv arguments; (ii) return zero when function debugging does not achieve an actual assembly return instruction; and (iii) detect and promote local to global variables when it is referenced in more than one function context.

A particular case to be handled by RevEngE are strings, whose memory addresses references must be lifted to ASCII. Whereas static decompilation approaches are able only to lift references to strings declared in `.static` binary sections, RevEngE can lift any string, including dynamically-generated ones, by runtime-dumping memory content and interpreting it as strings any time that a string declaration and/or reference is identified.

Introspection. RevEngE supports decompiling code which call external functions from linked libraries via an introspection procedure to identify function parameters and return values, as shown in Code 11, which allows RevEngE declaring variables from the correct type and returning the correct value via the `eax` register.

```
printf@stdio.h: int printf ( const char * format, ...
     ); (Return: int) (N_Args: 2)
```

Code Snippet 11: Introspection Procedure. External function prototypes are identified by searching for function and library names on the Internet and parsing them to a format suitable for RevEngE decompilation.

RevEngE's introspection procedure is inspired in previous approaches [39] to automate reverse engineering, thus automatically searching for function prototypes on the Internet based in the identified library which was being debugged when an introspection procedure was requested. The Internet search is completely automated via the Python's `googlesearch` [22] module, thus not requiring any analyst intervention. Parsed prototypes are locally stored in a `pickle` [41] blob to speed up further queries, since, for most applications, the same function calls are likely to appear many times along their execution. The reasoning for the adoption of the internet-based parser is to support even newly-created libraries. In the case where the libraries' prototypes are not found on the Internet, the analyst is prompted for manual intervention.

Analysis Passes. Before outputting high-level code, the IR can be analyzed to optimize the code to be generated and/or to discover bugs in the executed code. RevEngE implements an analysis step over the IR to discover reads from uninitialized memory positions. In such cases, RevEngE creates a new variable in the referred position and initialize the memory region with zeros an generates an uninitialized macro at the decompilation time, thus warning the analyst about the identified bug in the malware execution. RevEngE also implements an unused variable elimination by discarding variables associated to memory regions which were read only once and not further referred, thus reducing the overall amount of code to be inspected by the analyst.

Code Generation. RevEngE ensures the correct statement emission order by keeping a list a list of executed instruction addresses. This list is used to refer to a list that stores the objects corresponding to the instructions pointed by that address in the multiple context. This indirection is required since the same address may originate multiple, distinct objects according the context the instruction is called (e.g., the decompilation of the same function using distinct arguments). This indirection allows RevEngE to naturally perform loop unrolling during code emission. As RevEngE traverse the instruction addresses list, it pops the top of stack, thus serializing loops, as shown in Figure 3.

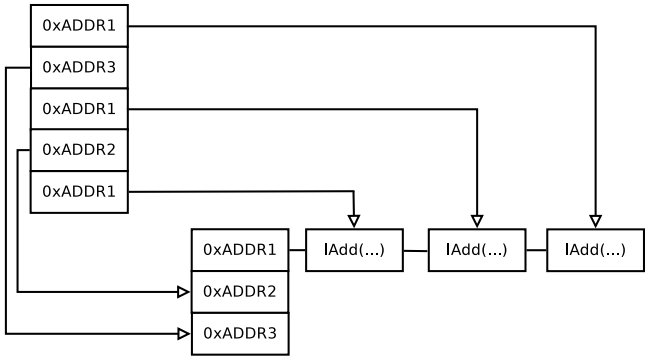

Figure 3: Code Generation. RevEngE keeps distinct objects for the same instruction address, thus representing the multiple calling contexts. Loop unrolling is performed by removing the top of stack each time a given instruction address is referred.

ii After the code emission step, gcc compiles the emitted code using gcc. RevEngE can also execute the compiled code and asserts whether the returned code matches the code returned to GDB by the original application.

5 EVALUATION

In this section, we evaluate our debug-oriented decompilation approach via the decompilation of real ELF malware samples using RevEngE. We evaluate: (i) whether RevEngE decompilation is correct or not; (ii) whether RevEngE actually helps analysts on decompiling obfuscated, real malware code; and (iii) whether RevEngE is

effective on reassembling decompiled functions in a new piece of high-level malware code.

TestBed. All tests were performed in an fresh Ubuntu 16.04 LTS installation, with RevEngE set on top of a standard GDB installation, in an network-isolated environment to prevent the debugged malware samples from infecting other hosts.

Dataset. To evaluate RevEngE, we reverse engineered the x86 ELF binaries available in the VirusShare repository [56]. Whereas ELF malware are available for multiple platforms and both and GDB can handle samples in a multi-platform manner, we limited our evaluation to x86 since it is the currently most widespread platform for malware dissemination [13].

Decompilation Correctness. RevEngE was developed using programming 101 code samples to assert decompilation correctness in comparison to the available source code. After we ensured the basic correctness of RevEngE decompilation procedures, we extended RevEngE evaluation to real, complex binary applications without source code availability. We considered 256 self-contained ELF binaries from the `/bin` and `/usr/bin` directories as goodware samples and 2,000 ELF binaries from the VirusShare dataset as malware samples.

To verify RevEngE capabilities on handling instructions present on real binaries, we set breakpoints on binaries' entry points (`main`) and `exit` points and decompiled all instructions between them. For all cases, we considered only the decompilation of instruction present in the binary itself and not on external libraries, since these are handled by the introspection procedures,

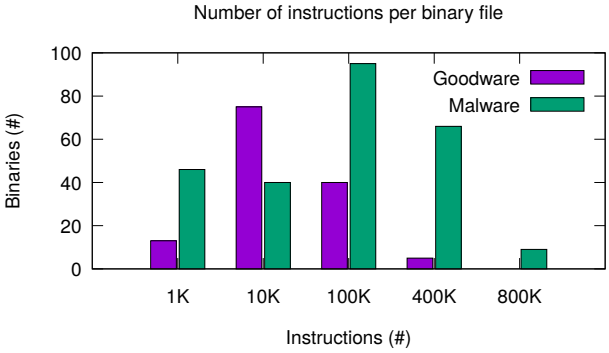

Figure 4: Number of instructions per binary. Malware samples executed more instructions than goodware samples.

Figure 4 shows the number of instructions executed during goodware and malware tracing procedures. We notice that, in an overall manner, malware samples executed more instructions than goodware samples, which can be explained by: (i) malware being more autonomous than goodware; and (ii) malware being more obfuscated than goodware. In the first case, some goodware samples finished their execution after only a hundred of instructions had been executed due to the lack of input parameters and/or arguments whereas most malware samples finished their execution normally since they relied on default parameters. In the latter, whereas no goodware sample was packed, many malware samples are packed and compressed, thus these samples spent significant processing time on unpacking routines, causing their traces to grow.

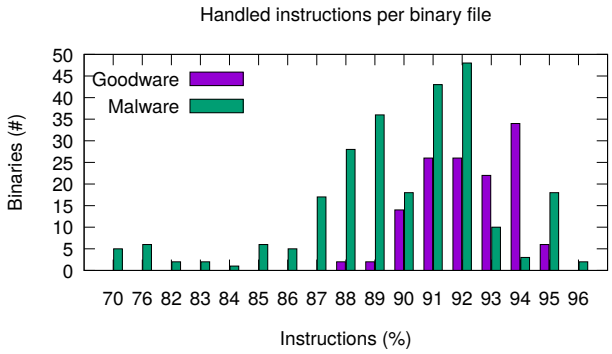

Figure 5: Handled instructions per binary. Most binaries were successfully handled. Malware samples impose greater challenges than goodware samples.

Figure 5 shows the number of binaries and the percentage of instructions that were successfully handled by RevEngE. We notice that, in an overall manner, RevEngE succeeds on handling most of binary instructions, even assuming a compact model of only handling integer operations. This result shows that handling external libraries via an introspection procedure is a successful strategy, since complex instructions (e.g., floating point) are much more present on libraries than on the binaries themselves and RevEngE does not need to handle the complexity of all instructions embedded by them. Despite the overall success, we notice that the smallest rates of instruction handling are observed for malware samples instead of goodware, since, as it is expected, some malware constructions frequently employed by malware are really challenging to be addressed by disassemblers and decompilers.

We reinforce that RevEngE's goal is not to decompile entire binaries, but slices of code defined by the analysts. Therefore, the obtained results (≈90% coverage) must be understood as a great probability of an analyst finding a slice of code that is successfully decompiled by RevEngE. In the cases where RevEngE is not able to lift an instruction, the analyst is prompted for manual intervention.

Malware de-obfuscation. As revealed by the previous experiments, a fraction of the considered malware samples is packed (e.g., 4% of all samples are packed with UPX), thus resulting in larger traces due to the execution of their unpacking routines. Whereas RevEngE has shown to be able to handle UPX code without stopping decompilation, packing makes understanding the final result a hard task, since unpacking code is mixed among the actual malware code that the malware is interested into, as shown in Code 12. A significant advantage of RevEngE's approach is that an analyst can step the unpack routine and start decompiling only after the unpacking routine has ended, thus avoiding mixing unpacking and malware code.

```
1 var117 = var115 << var97;
2 //Code page address discovered by the analyst
3 var118 = var117 & var103;
```

Code Snippet 12: Excerpt of UPX unpacking routine decompilation. Unpacking code mixed among malware decompilation code.

Malware Reassembly. To evaluate RevEngE's decompilation and reassembly capabilities, we reverse-engineered real malware samples, selected code regions implementing malicious behaviors described in a malware taxonomy [23], decompiled these code excerpts into new functions, and reassembled them into a new malware sample.

Code 13 shows the assembly instructions present in the function trace of a sample[2] from the Tsunami/Backdoor family [28] (67% detection, according VirusTotal).

```
call 0x8048dfc <rand@plt>
mov  %eax,%ecx mov    $0x66666667,%eax
imul %ecx      sar    %edx
mov  %ecx,%eax sar    $0x1f,%eax
sub  %eax,%edx mov    %edx,%eax
shl  $0x2,%eax
```

Code Snippet 13: Tsunami/Backdoor. Assembly code for the traced function.

Code 14 shows the decompilation result for `makestring` function. The function fills a string buffer by repeatedly attributing to each char a values attributed by generating a random number and performing a series of transformations over it. Random strings are used in malware samples, for instance, as infection markers and/or in fingerprinting procedures.

```
void makestring(char *var3) {
 int var1=0, var2=MAX_STRING,
 var6=0x666667, var9=0x1f, var12=2;
 for(var4=var1;var4<var2;var4++){
   var5=rand();      var7=var6/var5;
   var8=var6%var5;   var10=var7>>var9;
   var11=var8-var10; var13=var11<<var12;
   var3[var4]=var13;
```

Code Snippet 14: Tsunami/Backdoor. Decompiled code function.

Code 15 shows the assembly instructions present in the function trace of a sample[3] having Exploit/Trojan behavior (45% detection, according VirusTotal).

```
call 0x80484b4 <atoi@plt>
add  $0x10,%esp mov  %eax,%eax
mov  %eax,%eax  mov  %eax,-0x18(%ebp)
cmpl $0x2,0x8(%ebp)
jle  0x804862a <main+90>
push $0x1 call 0x80484a4 <exit@plt>
```

Code Snippet 15: Exploit/Trojan. Assembly code for the traced function.

Code 16 shows the decompilation result for `main` function. The function receives command line arguments, convert them to int and check if they match pre-defined values. This strategy is employed by malware samples to avoid analysis procedures, since the malicious behavior is exhibited only if the right parameters are passed, which happens in the infected machine, where the malicious binary is launched by a loader, but does not happen in analyst's machines, where the binary is launched in an standalone manner.

```
char var1[MAX_STRING];
 int var2=0, var3=3, var4=1,
 var6=0xf, var7=2, var8=0xff;
 if(argc==var3){  var5=atoi(argv[var4]);
  if(var5==var6){ var5=atoi(argv[var7]);
   if(var5==var8){
```

Code Snippet 16: Exploit/Trojan. Decompiled code function.

Code 17 shows the assembly instructions present in the function trace of a sample[4] from the Micmp/Backdoor family (34% detection, according VirusTotal).

```
call 0x8048734 <time@plt>
add  $0x4,%esp  push %eax
call 0x8048794 <srand@plt>
add  $0x10,%esp sub  $0x4,%esp
sub  $0xc,%esp  call 0x8048814 <rand@plt>
add  $0xc,%esp  mov  %eax,%edx
sar  $0x1f,%edx idiv %ecx
```

Code Snippet 17: Micmp/Backdoor. Assembly code for the traced function.

Code 18 shows the decompilation result for `return_randip` function. The function returns a random IP address by generating a large random number and dividing it into the four IPv4 octets. Random IP generation is used, for instance, in network scanning malware.

```
void return_randip(char *var1){
 int var3=0xB;  srand(time(NULL));
 var2 = rand(); var4 = var2 / var3;
 var5 = rand(); var6 = var5 / var3;
 var7 = rand(); var8 = var7 / var3;
 var9 = rand(); var10 = var9 / var3;
 sprintf(var1,"%d.%d.%d.%d",var...);
```

Code Snippet 18: Micmp/Backdoor. Decompiled code function.

Code 19 shows the assembly instructions present in the function trace of a sample[5] from the Small/Backdoor family [32] (62% detection, according VirusTotal).

```
movl $0x8049798,(%esp)
call 0x80487a8 <system@plt>
movl $0x80497bb,(%esp)
call 0x80487a8 <system@plt>
```

Code Snippet 19: Small/Backdoor. Assembly code for the traced function.

Code 20 shows the decompilation result for `open_firewall` function. The function executes commands to clean previous firewall rules and sets it to accept incoming connection, which is essential for a backdoor malware. The decompiled code shows RevEngE's capability of dynamically identifying parameters values, since the `iptables` rules are originally passed to the `system` via the `esp` stack pointer and RevEngE recovers the values by asking GDB to interpret the pointed address as a NULL-terminated string.

```
void open_firewall(){
 char var1[]="iptables -F INPUT";
 char var2[]="iptables -P INPUT ACCEPT";
 system(var1); system(var2);
```

[2] ffc7be26912b5aca63e55dc7c830f28a
[3] fb437621f3249c647c88350e068fd07
[4] ffb00447d40b0ae015752dd484d09de8
[5] 0a7e7a26796bf09112e997e2bd07ef24

Code Snippet 20: Small/Backdoor. Decompiled code function.

Code 21 shows the assembly instructions present in the function trace of a sample[6] from the RST/Virus family [53] (69% detection, according VirusTotal).

```
call 0x804a104 <openlog@plt>
push %ebx push $0x806f5e7 push $0x7
call 0x8049fa4 <syslog@plt>
call 0x804a1b4 <closelog@plt>
<userfile_remove>:
call   8049f54 <remove@plt>
```

Code Snippet 21: RST/Virus. Assembly code for the traced function.

Code 22 shows the decompilation result for debug function. The function checks if the system is monitored by `syslog`. This strategy is by malware samples to implement evidence removal procedures, by deleting the log files and thus remaining undetected. The decompiled code shows that RevEngE was able to recover not only the command string but also the parameter string (`"r"`) passed to the open function.

```
int debug(){
 FILE *var1; char var2[]="/var/log/syslog", var4[]="r"
      ;
 int var3=0; var1 = fopen(var2,var4);
 if(var1){ var3=1; }
 return var3;
```

Code Snippet 22: RST/Virus. Decompiled code function.

We reassembled the previously presented decompiled functions into a new malware sample. It performs the following actions: (i) It starts by checking the proper command line arguments to exhibit a malicious behavior only when the proper flags are inputted; (ii) checks for previous infection signs and creates a random strings to indicate the current infection; (iii) opens the firewall and sets a backdoor; (iv) spread itself to random IP addresses; and (v) finishes its execution by checking and removing log files.

The reassembled malware samples compiled properly, thus showcasing RevEngE's decompilation capabilities. The malware sample was submitted to VirusTotal and was not detected by any AV [57]. Therefore, we envision RevEngE's malware reassembly capabilities being used for fast PoCs prototyping for security evaluations, thus avoiding analysts to re-implement large portions of code.

6 DISCUSSION

We here present the implications of our proposed debug-oriented decompilation approach, including RevEngE advances and limitations, and existing development gaps.

Advances. RevEngE contributes to advance the current state of malware analysis development by proposing the adoption of a debug-centric decompilation approach. This approach benefits of analyst's expertise to overcome malware decompilation challenges and also benefits from the dynamic inspection capabilities of debugging solutions to analyze constructions which could only be resolved in runtime.

[6] 063e9fe33e46412f71f1d65a43fcfbfa

Why malware decompilation? Code decompilation is hard in most scenarios. However, malware decompilation exacerbates the existing challenges by frequently presenting "tricky" constructions that are hardly ever seem on benign software. On the other hand, malware samples have already been often handled via manual debugging procedures, which makes their decompilation a suitable case for RevEngE operation, thus our choice of malware decompilation to present our debug-oriented decompilation approach.

Limitations. Whereas outsourcing part of decompilation tasks to analysts is an effective measure, it might also be cost, since human analysts are an expensive and limited resources, which can be understood as a drawback of the debug-oriented decompilation approach regarding scenarios which require malware analysis at massive scales. For the future, we plan to investigate how to automate RevEngE operation in such scenarios.

Analysis Evasion. RevEngE is implemented on top of GDB, thus it is subject to many anti-analysis techniques targeting this debugger. We highlight, however, that RevEngE's goal is not to be the definitive malware decompiler solution, but a prototype to streamline the debug-oriented decompilation approach. We expect that the RevEngE concept could inspire other researchers when developing their decompilation solutions, either by relying on analyst's expertise, or by leveraging the innovative, real-time OOP implementation proposed by RevEngE.

Portability. RevEngE was implemented on top of GDB to benefit from its inspection capabilities. Whereas easing PoC development, the Linux platform present fewer malware samples than other ones, such as Windows. Thus, for the future, we envision porting RevEngE to other platforms, including as Windows, to broad our understanding about malware decompilation. RevEngE can be implemented in Windows, for instance, by leveraging Intel Pin [31], which allows tracking individual instruction in the same way as GDB does.

Open Decompilation challenges. Decompilation solutions present other open challenges in addition to the ones presented in Section 3, mainly regarding the development of new programming paradigms, such as Object-Oriented Programming (OOP) and parallel programming. When decompiling OOP code, decompilers would have to handle constructors and destructors as well as virtual methods. Whereas translating OOP code to structured code might be an alternative, since assembly code is not aware of high level constructs, it may result in undesirable artifacts being generated, as already observed in previous work [18]. Moreover, when decompiling parallel programs, the decompiler would have to be able to detect data transition/propagation across threads. In the malware context, threads might be used for attackers to distribute their payloads and a malware decompiler should be able to serialize the payload to provide a properly decompiled malware source code.

7 CONCLUSION

In this paper, we proposed a debug-oriented decompilation approach leveraging analyst's expertise to to overcome the decompilation challenges. We implemented RevEngE, the Reverse Engineering Engine for malware decompilation and reassembly, as a PoC for evaluating the proposed approach. RevEngE implements GDB extensions that intercept and introspect into executed functions to

build an Intermediate Representation (IR) in real-time, thus allowing decompilation to occur at any time. RevEngE was evaluated leveraging x86 ELF malware samples decompiled to a new malware sample composed of independent functions from five known malware samples. The resulting malware was not detected by any VirusTotal's AV although all original malware samples were detected by multiple AVs, thus showcasing that the debug-oriented decompilation proposal as a practical approach.

Reproducibility. The source code of the developed prototype was released as open source and it is available on `github`: https://github.com/marcusbotacin/Reverse.Engineering.Engine. An interactive version of RevEngE is available on the `corvus` platform: https://corvus.inf.ufpr.br/.

Acknowledgments. Marcus would like to thank the the Brazilian National Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process 164745/2017-3). Authors would like to thank the Coordination for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E. Porter. 2019. x86-64 Instruction Usage among C/C++ Applications. https://aakshintala.com/papers/instrpop-systor19.pdf.

[3] U. Bayer, C. Kruegel, and E. Kirda. 2006. TTAnalyze: A tool for analyzing malware. In *15th European Inst. for Comp. Antivirus Research (EICAR 2006) Annual Conf.* EICAR.

[4] David Binkley, Nicolas Gold, and Mark Harman. 2007. An Empirical Study of Static Program Slice Size. *ACM Trans. Softw. Eng. Methodol.* 16, 2, Article 8 (April 2007). https://doi.org/10.1145/1217295.1217297

[5] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. 2015. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 745–756. https://doi.org/10.1145/2810103.2813627

[6] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti- VM Technologies. http://www.kernelhacking.com/rodrigo/docs/blackhat2012-paper.pdf.

[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469. http://dl.acm.org/citation.cfm?id=2032305.2032342

[8] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babi ć, and Dawn Song. 2010. Input Generation via Decomposition and Re-stitching: Finding Bugs in Malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, New York, NY, USA, 413–425. https://doi.org/10.1145/1866307.1866354

[9] G. Canfora, A. Cimitile, and M. Munro. 1994. RE2: Reverse-engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice* 6, 2 (1994), 53–72. https://doi.org/10.1002/smr.4360060202

[10] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 99–116. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua

[11] Cristina Cifuentes. [n. d.]. Reverse Compilation Techniques. https://yurichev.com/mirrors/DCC_decompilation_thesis.pdf.

[12] Cristina Cifuentes, Trent Waddington, and Mike Van Emmerik. 2001. Computer Security Analysis Through Decompilation and High-Level Debugging. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01) (WCRE '01)*. IEEE Computer Society, Washington, DC, USA, 375–. http://dl.acm.org/citation.cfm?id=832308.837157

[13] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti. 2018. Understanding Linux Malware. In *2018 IEEE Symposium on Security and Privacy (SP)*. 161–175. https://doi.org/10.1109/SP.2018.00054

[14] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. 15th ACM Conf. Computer and Comm. Security (CCS '08)*. 51–62.

[15] Felix Engel, Rainer Leupers, Gerd Ascheid, Max Ferger, and Marcel Beemster. 2011. Enhanced Structural Analysis for C Code Reconstruction from IR Code. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems (SCOPES '11)*. ACM, New York, NY, USA, 21–27. https://doi.org/10.1145/1988932.1988936

[16] Julien et al. [n. d.]. Next generation debuggers for reverse engineering. http://s.eresi-project.org/inc/articles/bheu-eresi-article-2007.pdf.

[17] Thomas Dullien et al. [n. d.]. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. https://static.googleusercontent.com/media/www.zynamics.com/pt-BR//downloads/csw09.pdf.

[18] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE '11)*. IEEE Computer Society, Washington, DC, USA, 347–356. https://doi.org/10.1109/WCRE.2011.49

[19] Jose Manuel Rios Fonseca. [n. d.]. Interactive Decompilation. http://paginas.fe.up.pt/~mei04010/thesis.pdf.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[21] gef. [n. d.]. GEF - GDB Enhanced Features for exploit devs & reversers. https://github.com/hugsy/gef.

[22] GoogleSearch. 2018. GoogleSearch. https://pypi.org/project/google-search/.

[23] André Ricardo Abed Grégio, Vitor Monte Afonso, Dario Simões Fernandes Filho, Paulo Lício de Geus, and Mario Jino. 2015. Toward a Taxonomy of Malware Behaviors. *Comput. J.* 58, 10 (07 2015), 2758–2777. https://doi.org/10.1093/comjnl/bxv047 arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/58/10/2758/5096654/bxv047.pdf

[24] Ilfak Guilfanov. [n. d.]. Decompilers and beyond. https://www.hex-rays.com/products/ida/support/ppt/decompilers_and_beyond_white_paper.pdf.

[25] Clifford R. Hollander. 1974. A Syntax-directed Approach to Inverse Compilation. In *Proceedings of the 1974 Annual ACM Conference - Volume 2 (ACM '74)*. ACM, New York, NY, USA, 750–750. https://doi.org/10.1145/1408800.1408926

[26] Barron C. Housel and Maurice H. Halstead. 1974. A Methodology for Machine Language Decompilation. In *Proceedings of the 1974 Annual Conference - Volume 1 (ACM '74)*. ACM, New York, NY, USA, 254–260. https://doi.org/10.1145/800182.810410

[27] ISECLAB. 2010. Anubis - Malware Analysis for Unknown Binaries. https://anubis.iseclab.org/.

[28] Kaspersky. 2010. Backdoor.Linux.Tsunami. https://threats.kaspersky.com/en/threat/Backdoor.Linux.Tsunami/.

[29] Daniel Kästner and Stephan Wilhelm. 2002. Generic Control Flow Reconstruction from Assembly Code. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES/SCOPES '02)*. ACM, New York, NY, USA, 46–55. https://doi.org/10.1145/513829.513839

[30] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. 2011. The Power of Procrastination: Detection and Mitigation of Execution-stalling Malicious Code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/2046707.2046740

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[32] Microsoft. 2007. Backdoor:Linux/Small. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Backdoor:Linux/Small.

[33] Jerome Miecznikowski and Laurie J. Hendren. 2002. Decompiling Java Bytecode: Problems, Traps and Pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, UK, 111–127. http://dl.acm.org/citation.cfm?id=647478.727938

[34] A. Moser, C. Kruegel, and E. Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 421–430. https://doi.org/10.1109/ACSAC.2007.21

[35] newlog. [n. d.]. Radare2MSDN. https://github.com/newlog/r2msdn.

[36] Kenneth Oksanen. 2011. Detecting Algorithms Using Dynamic Analysis. In *Proceedings of the Ninth International Workshop on Dynamic Analysis (WODA '11)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2002951.2002953

[37] Roberto Paleari, Lorenzo Martignoni, Emanuele Passerini, Drew Davidson, Matt Fredrikson, Jon Giffin, and Somesh Jha. 2010. Automatic Generation

of Remediation Procedures for Malware Infections. In *USENIX Sec.* 1. http://dl.acm.org/citation.cfm?id=1929820.1929856
[38] PEDA. [n. d.]. PEDA - Python Exploit Development Assistance for GDB. https://github.com/longld/peda.
[39] Mario Polino, Andrea Scorti, Federico Maggi, and Stefano Zanero. 2015. Jackdaw: Towards Automatic Reverse Engineering of Large Datasets of Binaries. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Magnus Almgren, Vincenzo Gulisano, and Federico Maggi (Eds.). Springer International Publishing, Cham, 121–143.
[40] Pwndbg. [n. d.]. Pwndbg. https://github.com/pwndbg/pwndbg.
[41] Python. 2018. pickle - Python object serialization. https://docs.python.org/3/library/pickle.html.
[42] radare. 2019. radare. https://rada.re/.
[43] rdbv. 2017. Translator from ASM to C, but not decompiler. Something between compiler and decompiler. https://github.com/rdbv/cisol.
[44] Ed Robbins, Andy King, and Tom Schrijvers. 2016. From MinX to MinC: Semantics-driven Decompilation of Recursive Datatypes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 191–203. https://doi.org/10.1145/2837614.2837633
[45] Gabriel Rodríguez, José M. Andión, Mahmut T. Kandemir, and Juan Touriño. 2016. Trace-based Affine Reconstruction of Codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 139–149. https://doi.org/10.1145/2854038.2854056
[46] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. 2013. Native x86 Decompilation Using Semantics-preserving Structural Analysis and Iterative Control-flow Structuring. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 353–368. http://dl.acm.org/citation.cfm?id=2534766.2534797
[47] Maxime Serrano. 2013. Lecture Notes on Decompilation. https://www.cs.cmu.edu/~fp/courses/15411-f13/lectures/20-decompilation.pdf.
[48] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.
[49] snowman. 2019. snowman. https://derevenets.com/.
[50] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1
[51] Murugiah Souppaya and Karen Scarfone. 2013. Guide to Malware Incident Prevention and Handling for Desktops and Laptops. https://tinyurl.com/kh4mnjv.
[52] Greg Stitt and Frank Vahid. 2008. Binary Synthesis. *ACM Trans. Des. Autom. Electron. Syst.* 12, 3, Article 34 (May 2008), 30 pages. https://doi.org/10.1145/1255456.1255471
[53] TrendMicro. 2002. ELF_RST.B. https://www.trendmicro.com/vinfo/us/threat-encyclopedia/archive/malware/elf_rst.b.
[54] K. Troshina, A. Chernov, and A. Fokin. 2009. Profile-based type reconstruction for decompilation. In *2009 IEEE 17th International Conference on Program Comprehension*. 263–267. https://doi.org/10.1109/ICPC.2009.5090054
[55] Michael James van Emmerik. [n. d.]. Static Single Assignment for Decompilation. https://yurichev.com/mirrors/vanEmmerik_ssa.pdf.
[56] VirusShare. 2019. VirusShare. https://virusshare.com/.
[57] VirusTotal. 2019. Detection Results. https://tinyurl.com/yxdptfxn.
[58] Mark Weiser. 1984. Program Slicing. *IEEE Trans. Softw. Eng.* 10, 4 (July 1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248
[59] Maria F. Weller. 1974. A Pragmatic Look at Decompilers. In *Proceedings of the 1974 Annual ACM Conference - Volume 2 (ACM '74)*. ACM, New York, NY, USA, 753–753. https://doi.org/10.1145/1408800.1408930
[60] C. Willems, T. Holz, and F. Freiling. 2007. Toward automated dynamic malware analysis using cwsandbox. *IEEE Sec. & Priv.* 5 (2007). Issue 2.
[61] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. 2016. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *2016 IEEE Symposium on Security and Privacy (SP)*. 158–177. https://doi.org/10.1109/SP.2016.18
[62] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2013. Obfuscation Resilient Binary Code Reuse Through Trace-oriented Programming. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 487–498. https://doi.org/10.1145/2508859.2516664
[63] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *2015 IEEE Symp. on Security and Privacy*. 55–69.
[64] J. Zhang, R. Zhao, and J. Pang. 2007. Parameter and Return-value Analysis of Binary Executables. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Vol. 1. 501–508. https://doi.org/10.1109/COMPSAC.2007.163
[65] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. 2005. Experimental Evaluation of Using Dynamic Slices for Fault Location. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging (AADEBUG'05)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/1085130.1085135