Helping Johnny to Analyze Malware

A Usability-Optimized Decompiler and Malware Analysis User Study

Khaled Yakdan*†, Sergej Dechand*, Elmar Gerhards-Padilla†, Matthew Smith*
*University of Bonn, Germany
{yakdan, dechand, smith}@cs.uni-bonn.de
†Fraunhofer FKIE, Germany
elmar.gerhards-padilla@fkie.fraunhofer.de

***Abstract*—Analysis of malicious software is an essential task in computer security; it provides the necessary understanding to devise effective countermeasures and mitigation strategies. The level of sophistication and complexity of current malware continues to evolve significantly, as the recently discovered "Regin" malware family strikingly illustrates. This complexity makes the already tedious and time-consuming task of manual malware reverse engineering even more difficult and challenging. Decompilation can accelerate this process by enabling analysts to reason about a high-level, more abstract from of binary code. While significant advances have been made, state-of-the-art decompilers still produce very complex and unreadable code and malware analysts still frequently go back to analyzing the assembly code.**

In this paper, we present several semantics-preserving code transformations to make the decompiled code more readable, thus helping malware analysts[1] understand and combat malware. We have implemented our optimizations as extensions to the academic decompiler DREAM. To evaluate our approach, we conducted the first user study to measure the quality of decompilers for malware analysis. Our study includes 6 analysis tasks based on real malware samples we obtained from independent malware experts. We evaluate three decompilers: the leading industry decompiler Hex-Rays, the state-of-the-art academic decompiler DREAM, and our usability-optimized decompiler DREAM^{++}. The results show that our readability improvements had a significant effect on how well our participants could analyze the malware samples. DREAM^{++} outperforms both Hex-Rays and DREAM significantly. Using DREAM^{++} participants solved $3\times$ more tasks than when using Hex-Rays and $2\times$ more tasks than when using DREAM.

I. INTRODUCTION

The analysis of malware is a fundamental problem in computer security. It provides the necessary detailed understanding of the functionality and capabilities of malware, and thus forms the basis for devising effective countermeasures and mitigation strategies. Created by professional and highly skilled adversaries, modern malware is increasingly sophisticated and complex. Advanced malware families such as Stuxnet [26], Uroburos [29], and Regin [50] are examples of the level of sophistication and complexity of current malware. These malware families show the extraordinary lengths malware authors go to to conceal their activities and make the already tedious and time-consuming task of manual reverse engineering of malware even more challenging and difficult.

Due to the inability of a program to identify non-trivial properties of another program, the generic problem of automatic malware analysis is undecidable [44]. As a result of this limitation, security research has focused on automatically analyzing specific types of functionality, such as the identification of cryptographic functions [13], automatic protocol reverse engineering [12, 53], and the detection of DGA-based malware [2]. As another result of this limitation, security analysts often have to resort to manual reverse engineering for detailed and thorough analysis of malware, a difficult and time-consuming process. As a remedy, security researchers have started to explore approaches that assist analysts during analysis instead of replacing them. The proposed methods accelerate the analysis process by correctly identifying functions in binaries [4, 47], reliably extracting binary code [7, 35, 37, 42, 58], deobfuscating obfuscated executable code [20, 56], and recovering high-level abstractions from binary code through decompilation [45, 57].

Decompilation offers an attractive method to assist malware analysis by enabling analyses to be performed on a high-level, more abstract form of the binary code. At a high level, decompilation consists of a collection of *abstraction recovery* mechanisms to recover high-level abstractions that are not readily available in the binary code. Both manual and automated analyses can then be performed on the decompiled program code, reducing both the time and effort required. Towards this goal, the research community has addressed principled methods for recovering high-level abstractions required for source code reconstruction. This includes approaches for recovering data types [39, 40, 48] and high-level control-flow structure such as `if-then-else` constructs and `while` loops [45, 57] from binary code. While significant advances have been made, state-of-the-art decompilers still create very complex code and do not focus on readability.

In this paper, we argue that a human-centric approach can significantly improve the effectiveness of decompilers. To this end, we present several semantics-preserving code transformations to simplify the decompiled code. Improved readability makes the decompiled code easier to understand and thus can accelerate manual reverse engineering of malware. The key insight of our approach is that the abstractions recovered

[1]While some people have come to interpret Johnny as a derogatory name in usability studies, the name was never meant as such. So in this study malware analysts are our Johnnies and are highly skilled people.

during previous decompilation stages can be leveraged to devise powerful optimizations. The main intuition driving these optimizations is that the decompiled code is easier to understand if it can be formed in a way that is similar to what a human programmer would write. Based on this intuition, we devise optimizations to simplify expressions and control-flow structure, remove redundancy, and give meaningful names to variables based on how they are used in code. Also, we develop a generic query and transformation engine that allows analysts to easily write code queries and transformations. We have implemented our usability extensions on top of the state-of-the-art academic decompiler DREAM [57]. We call this extended version DREAM^{++}.

While a lot of work has been done on improving decompilers, the evaluation of these approaches has never included user studies to validate if the optimizations actually help real malware analysts. Cifuentes et al.'s pioneering work [16] and numerous subsequent works [15, 17, 24, 27, 45, 57] all evaluated the decompiler quality based on some machine-measurable readability metric such as the number of `goto` statements in the decompiled code or how much smaller the decompiled code was in comparison to the input assembly. Moreover, a significant amount of previous work featured a manual qualitative evaluation on a few, small, sometimes self-written, examples [16, 24, 27, 28].

In this paper, we present the first user study on malware analysis. We conducted a study with 21 students who had completed a course on malware analysis and 9 professional malware analysts. The study included 6 reverse engineering tasks of real malware samples that we obtained from independent malware experts. The results of our study show that our improved decompiler DREAM^{++} produced significantly more understandable code and outperformed both the leading industry and academic decompilers: Hex-Rays [30] and DREAM. Using DREAM^{++} participants solved $3\times$ more tasks than when using Hex-Rays and $2\times$ more tasks than when using DREAM. Both experts and students rated DREAM^{++} significantly higher than the competition.

In summary, we make the following contributions:

- *Usability extensions to decompiler*. We present several semantics-preserving code transformations to simplify and improve the readability of decompiled code. Our optimizations leverage the high-level abstractions recovered by previous decompilation stages. We have implemented our techniques as extensions to the state-of-the-art academic decompiler DREAM [57]. We call the extended decompiler DREAM^{++}.
- *New usability metric to evaluate decompiler quality*. We propose to include the *human factor* in a metric to evaluate how useful a decompiler is for malware analysis. Although previous work has proposed several quantitative metrics to measure decompiler quality, surprisingly the human factor has not yet been studied. Given that manual reverse engineering is a main driving factor behind decompilation research, this is a serious oversight and we hope our approach will also serve as a benchmark for future research.
- *Evaluation with user study*. We conduct the *first* user study to evaluate the quality and usefulness of decompilers for malware analysis. We conduct our study both with students trained in malware analysis as well as professional malware analysts. The results provide a statistically significant evidence that DREAM^{++} outperforms both the leading industry decompiler Hex-Rays and the original DREAM decompiler in the amount of tasks successfully analyzed.

II. PROBLEM STATEMENT & OVERVIEW

The focal point of this paper is on improving the readability of decompiler-created code to accelerate the analysis of malware. Code readability is essential for humans to correctly understand the functionality of code [10]. We conducted several informal interviews with malware analysts to identify shortcomings of state-of-the-art decompilers that negatively impact readability. We also conducted cognitive walkthroughs stepping through the process of restructuring malware code produced by Hex-Rays and DREAM to see what the problems of these two decompilers are. We group the discovered problems into three categories

1) Complex expressions: State-of-the-art decompilers often produce overly complex expressions. Such expressions are rarely found in source code written by humans and are thus hard to understand. This includes

a) Complex logic expressions: Logic expressions are used inside control constructs (e.g., `if-then` or `while` loops) to decide the next code to be executed. Complex logic expressions make it difficult to understand the checks performed in the code and the decisions taken based on them.

b) Number of variables: Decompiled code often contains too many variables. This complicates analysis since one must keep track of a large number of variables. Although decompilers apply a dead code elimination step, they still miss opportunities to remove redundant variables. In many scenarios, several variables can be merged into a single variable while preserving the semantics of the code.

c) Pointer expressions: Array access operations are usually recovered as dereference expressions involving pointer arithmetic and cast operations. Moreover, accesses to arrays allocated on the stack are recovered as expressions using the address-of operator (e.g., `*(&v + i)`).

We present our approach to tackle these problems in Section III.

2) Convoluted control flow: The readability of a program depends largely upon the simplicity of its sequencing control [23]. Two issues often complicates the control flow structure recovered by decompilers

a) Duplicate/inlined code: Binary code often contains duplicate code blocks. This usually results from macro expansion and function inlining during compilation. As a result, analysts may end up analyzing the same code block several times.

```
1  void *__cdecl sub_10006390(){
2    __int32 v13; // eax@14
3    int v14; // esi@15
4    unsigned int v15; // ecx@15
5    int v16; // edx@16
6    char *v17; // edi@18
7    bool v18; // zf@18
8    unsigned int v19; // edx@18
9    char v20; // dl@21
10   char v23; // [sp+0h] [bp-338h]@1
11   int v30; // [sp+30Ch] [bp-2Ch]@1
12   __int32 v36; // [sp+324h] [bp-14h]@14
13   int v37; // [sp+328h] [bp-10h]@1
14   int i; // [sp+330h] [bp-8h]@1
15   // [...]
16   v30 = *"qwrtpsdfghjklzxcvbnm";
17   v37 = *"eyuioa";
18   // [...]
19   v14 = 0;
20   v15 = 3;
21   if ( v13 > 0 )
22   {
23     v16 = 1 - &v23;
24     for ( i = 1 - &v23; ; v16 = i )
25     {
26       v17 = &v23 + v14;
27       v19 = (&v23 + v14 + v16) & 0x80000001;
28       v18 = v19 == 0;
29       if ( (v19 & 0x80000000) != 0 )
30         v18 = ((v19 - 1) | 0xFFFFFFFE) == -1;
31       v20 = v18 ? *(&v37 + dwSeed / v15 % 6)
32                 : *(&v30 + dwSeed / v15 % 0x14);
33       ++v14;
34       v15 += 2;
35       *v17 = v20;
36       if ( v14 >= v36 )
37         break;
38     }
39   }
40   // [...]
41 }
```

(a) Hex-Rays

```
1  LPVOID sub_10006390(){
2    int v1 = *"qwrtpsdfghjklzxcvbnm";
3    int v2 = *"eyuioa";
4    // [...]
5    int v18 = 0;
6    int v19 = 3;
7    if(num > 0){
8      do{
9        char * v20 = v18 + (&v3);
10       int v21 = v18 + 1;
11       int v22 = v21;
12       int v23 = v21 & 0x80000001L;
13       bool v24 = !v23;
14       if(v23 < 0)
15         v24 = !(((v23 - 1) | 0xfffffffeL) + 1);
16       char v25;
17       if(!v24)
18         v25 = *(((dwSeed / v19) % 20) + (&v1));
19       else
20         v25 = *(((dwSeed / v19) % 6) + (&v2));
21       v18++;
22       v19 += 2;
23       *v20 = v25;
24     }while(v18 < num);
25   }
26   // [...]
27 }
```

(b) DREAM

```
1  LPVOID sub_10006390(){
2    char * v1 = "qwrtpsdfghjklzxcvbnm";
3    char * v2 = "eyuioa";
4    // [...]
5    int v13 = 3;
6    for(int i = 0; i < num; i++){
7      char v14 = i % 2 == 0 ? v1[(dwSeed / v13) % 20]
8                            : v2[(dwSeed / v13) % 6];
9      v13 += 2;
10     v3[i] = v14;
11   }
12   // [...]
13 }
```

(c) DREAM^{++}

Fig. 1: Excerpt from the decompiled code of the domain generation algorithm of the Simda malware family. This example shows the main loop where the domain names are generated. At a high level, letters are picked at random from two arrays. Choosing the array from which to copy a letter is based on whether the loop counter is even or odd.

b) Complex loop structure: Control-flow structuring algorithms used by decompilers recognize loops by analyzing the control flow graph. For this reason, they recover the structure produced by the compiler which is optimized for efficiency but not readability. Stopping at this stage prevents decompilers from recovering more readable forms of loops as those seen in the source code written by humans.

We address these problems in Section V. At the core of our optimization is our code query and transformation framework which we describe in Section IV.

3) Lack of high-level semantics: High-level semantics such as variable names are lost during compilation and cannot be recovered by decompilers. For this reason, decompilers usually assign default names to variables. Also, some constants have a special meaning in a given context, e.g., used by an API function or as magic numbers for file types. In Section VI, we describe several techniques to give variables and constants meaningful names based on how they are used in the code.

As an example illustrating these problems, we consider the code shown in Figure 1. This Figure shows the decompiled code of the domain generation algorithm (DGA) of the Simda malware family produced by three decompilers: Hex-Rays (Figure 1a), DREAM (Figure 1b), and our improved decompiler DREAM^{++} (Figure 1c). Here, due to space restrictions, we only show the main loop where the domains are computed[2]. As shown in the snippets, the code produced by Hex-Rays and DREAM is rather complex and hard to understand. In the code produced by Hex-Rays, the loop variable `i` is never used inside the loop and the loop ends with a `break` statement. Moreover, the recovered checks for the parity of the loop counter involves complex low-level expressions (lines 26-30). Accessing the `char` arrays (`v37` and `v30`) uses pointer arithmetic, address-of operators, and dereference operators.

[2]The complete code can be found in the supplemental document https://net.cs.uni-bonn.de/fileadmin/ag/martini/Staff/yakdan/sm-oakland-2016.tar.gz

DREAM produced a slightly more readable code but still has a number of issues. Here, the recovered loop structure is not optimal and can be further simplified. Since the initial value of `v18` is zero, the condition of the `if` statement and the enclosed `do-while` loop are identical at the first iteration. This opens up the possibility to transform the whole construct into a more readable `while` loop.

Finally, the optimizations developed during the course of this paper further reduce the complexity of the code. As can be seen from Figure 1c, the code contains a simple `for` loop with a clear initialization step, condition, and increment step. With each loop iteration, a letter is selected from two `char` arrays (`v1` and `v2`) depending on the parity of the loop counter (`i % 2 == 0`) and the result is stored in the output array (`v3`).

Scope. DREAM^{++} is based on the DREAM decompiler which uses IDA Pro [33] to disassemble the binary code and build the control-flow graph of functions in the binary. Arguably, the resulting disassembly is not perfect and may contain errors if the binary is deliberately obfuscated. For the scope of this paper, we assume that the assembly provided to the decompiler is correct. Should the binary code be obfuscated, tools such as [7, 37, 58] can be used to extract the binary code. Furthermore, recent approaches such as [20, 56] can be used to deobfuscate the binary code before providing it as input to the decompiler.

A high-level overview of our approach is as follows. First, the binary file is decompiled using DREAM. This stage decompiles each function and generates the corresponding control flow graph (CFG) and the abstract syntax tree (AST). Each node in the AST represents a statement or an expression in DREAM's intermediate representation (IR). Our work starts here. We develop three categories of semantics-preserving code transformations to simplify the code and increase readability. These categories are *expression simplification*, *control-flow simplification* and *semantics-aware naming*. In the following sections, we discuss our optimizations in detail.

III. EXPRESSION SIMPLIFICATION

In this section, we present our optimizations to simplify expressions and remove redundancy from decompiled code.

A. Congruence Analysis

Congruence analysis is our approach to remove redundant variables from the decompiled code. The key idea is to identify variables that represent the same value and can be replaced by a single representative variable while preserving semantics. We denote such variables as *congruent variables*. DREAM already performs several optimizations to remove redundancy such as expression propagation and dead code elimination. However, there exist scenarios where traditional dead code elimination algorithms cannot remove redundant code. A prominent example is when compilers emit instructions to temporarily save some values that are later restored for further use. Depending on the control structure, this may result in circular dependency between the corresponding variables in the decompiler IR, preventing dead code elimination from removing them. As an example illustrating these scenarios, we consider the code

```
1  int foo(){
2    int x = bar();
3    int z = 10;
4    int y = x;
5    while(x < z){
6      z = qux();
7      x = y;
8    }
9    y = y + z
10   return y;
11 }
```

(a) Example code.

```
1  int foo(){
2    int v = bar();
3    int z = 10;
4    while(v < z){
5      z = qux();
6    }
7    v = v + z;
8    return v;
9  }
```

(b) Optimized code.

Fig. 2: Congruence Analysis

sample shown in Figure 2a. In this example, lines 4 and 7 copy a value between variables `x` and `y`. Also, replacing `x` and `y` by a single representative, e.g., variable `v`, does not change the semantics of the program. Moreover, this replacement results in two trivial copy statements of the form `v = v` (lines 4 and 7) which can be safely removed, resulting in the more compact and readable code shown in Figure 2b.

This simple example gives insight into the different properties of code that play a role in the characterization of variable congruence. In summary, the following aspects need to be covered.

1) *Same Types*: Congruent variables have the same data types. This requirement is necessary to avoid the changing semantics because of implicit type conversions. For example, the transformation would not be semantics-preserving if y was of type `short`.
2) *Non-Interfering Definitions*: Replacing congruent variables by a single representative does not change the definitions that reach program points where these variables are used. Note that this does not require that the live ranges of congruent variables do not interfere. For example, the definition of `x` at line 7 is located in the live range of `y`, i.e., between a definition of `y` (line 4) and a corresponding use of `y` (line 9). However, the definition is a simple copy statement `x = y` and therefore using any of `x` or `y` at line 9 preserves semantics.
3) *Congruence-Revealing Statements*: The previous checks are enough to guarantee the semantics-preserving property of unifying variables. However, applying this to all variables without limitation may negatively impact readability. Two non-interfering variables of the same type may have different semantics (e.g., one integer variable used as a loop counter and a second integer used as the size of a buffer). Not merging such variables enables us to give each of them a representative name based on how the variable is used in code. Based on that we limit congruence analysis to variables for which the code contains indications that they are used for the same purpose. That is, we only check variables involved in copy statements of the form `x = y`, which we denote as *congruence-revealing* statements.

At the core of these checks is information about liveness of variables. To this end, we perform a fixed-point intraprocedural

Algorithm 1 Congruence analysis

```
1:  procedure MergeCongruentVars(V)
2:      for (a, b) ∈ Candidatates(V) do
3:          if Congruent(a, b) then
4:              v ← Unify(a, b)
5:              V ← V \ {a, b}
6:              V ← V ∪ {v}
7:              UpdateLiveness(v)
8:  procedure Congruent(a, b)
9:      return InterfenceFree(a, b) ∧ InterfenceFree(b, a)
10: procedure InterfenceFree(x, y)
11:     for all d ∈ Def(y) do
12:         if d ∈ LiveRange(x) ∧ d ≢ y = x then
13:             return false
14:     return true
```

live variable analysis, a standard problem from compiler design [41, p. 443]. At a high level, live variable analysis determines which variables are *live* at each point in the program. A variable v is live at a particular point in the program $p \in P$ if p is located on an execution path from a definition of v and a use of v that does not contain a redefinition of v. This set of program points constitute the *live range* of the variable.

$$\text{LiveRange}(v) = \{p \in P : v \text{ live at } p\}$$

Algorithm 1 implements this idea by first calculating the set of candidate variable pairs, i.e., variables of the same types that are involved in congruent-revealing statements, and then checking these pairs for congruence. For each candidate pair (x, y), the algorithm checks if they do not have interfering definitions. In particular, the procedure InterferenceFree checks if each definition of variable y is either not located in the live range of x, or it is a copy statement of the form $y = x$. The same check is also done at the definitions of x. When two congruent variables are identified, the procedure Unify 1) chooses one representative variable v; 2) replaces all occurrences of the concurrent variables by the representative; and 3) removes the trivial copy statements resulted from this unification (of the form `v = v`). Next, the set of variables V is updated. Finally, liveness information of the newly added variable is updated as follows:

$$\text{LiveRange}(v) = \text{LiveRange}(x) \cup \text{LiveRange}(y)$$

Not that we do not require that congruent variables must have the same values at all program points. They may have different values at points where their live ranges do not interfere. For example, although different values of variables `x` and `y` reach the return statement in the code shown in Figure 2a, `x` is not live at lines 9 and 10. This enables us to use the same variable for both `x` and `y`.

B. Condition Simplification

The goal of this step is to find the simplest high-level form of logic expressions in the decompiled code. These expressions are very important for understanding the control flow of a program since they are used in control statements, such as `if-then-else` statements or `while` loops, to decide what code to execute next. Simplifying logic expressions is helpful in two aspects: first, it helps to recover the semantically equivalent high-level conditions to the low-level checks emitted by the compiler. Second, it helps to clear any misunderstanding caused by errors in the original code.

```
1  [...]
2  result = GetVersionExW(&VersionInformation);
3  if(result
4     && VersionInformation.dwPlatformId == 2
5     && ( VersionInformation.dwMajorVersion >= 5
6        || VersionInformation.dwMajorVersion <= 6))
7  [...]
```

(a) Hex-Rays

```
1  [...]
2  BOOL result = GetVersionExW(&VersionInformation);
3  if(result != 0
4     && VersionInformation.dwPlatformId == 2)
5  [...]
```

(b) Dream^{++}

Fig. 3: Excerpt from the decompiled code from a Stuxnet sample. The code checks the version of the Windows operating system.

Low-level checks. During compilation a compiler uses a transformation called *tiling* to reduce the high-level program statements into assembly statements. As a result, each high-level statement can be transformed into a sequence of semantically equivalent assembly instructions. During this process, high-level predicates are transformed to semantically equivalent low-level checks that can be executed efficiently. As an example, we consider the code shown in Figure 1a. The right-hand side of the assignment at line 30 is a complex expression that checks whether the variable `v19` is an even or odd number. This does not look like a common operation used in source code, but it is equivalent to the high-level operation of computing `v19 % 2 == 0`.

Errors in the code. Malware code may contain logic errors that can create confusion for analysts. Malware analysts assume that the code they analyze performs some meaningful task they need to find out. They also know that malware often uses several tricks to hide its true functionality. With this mindset, when analysts observe a seemingly useless code, they need to double-check in order to exclude the possibility of a trick aimed at making the code looks useless. As a result, some time is wasted. The simple example from the Stuxnet malware family shown in Figure 3a illustrates this case. This code checks the version of the Windows operating system, a common procedure in *environment-targeted malware* [55]. However, the OR expression (marked in red) is always satisfied; any integer is either bigger than 5 or smaller than 6. Most probably, the malware authors intended to use an AND expression instead but did not for some reason. Simplifying this expression results in the code shown in Figure 3b.

To provide a generic simplification approach, we base our techniques on the `Z3` theorem prover [21]. Our approach proceeds as follows. First, we transform logic expressions in the DREAM IR into semantically equivalent symbolic expressions for the `Z3` theorem prover. To achieve a faithful representation, we model variables as fixed-size bit-vectors depending on their types. The theory of bit-vectors allows modeling the precise semantics of unsigned and signed two-complements arithmetic. During this transformation, we keep a mapping between each symbolic variable and the corresponding variable it represents in the original logic expression. Second, we use the theorem prover to simplify and normalize the symbolic expressions. Finally, we use the mapping to construct the simplified version of the logic expression in DREAM IR.

C. Pointer Transformation

Accessing values through pointer dereferencing using pointer arithmetic can be confusing. Also, accessing buffers allocated on the stack may result in convoluted decompiled code that is difficult to understand.

Pointer-to-array transformation. Here, we use the observation that in C a pointer can be indexed like an array name. This representation clearly separates the pointer variable from the expression used to compute the offset from the start address. To guarantee the semantics-preserving property of this transformation, we search for variables of pointer types that are accessed consistently in the code. That is, all data that is read or written using the pointer variable have the same type τ. In this case, dereferencing these variables can be represented as array with elements of type τ. Here the resulting offset expression must be adjusted according to the size of type τ. For example, if a pointer `p` is consistently used to access 4-byte integers, then expressions such as `*(p + 4 * i)` can be transformed into the more readable `p[i]` form.

Reference-to-pointer transformation. In this step, we transform variables that are only used in combination with *address-of operator* (`&`) into pointer variables. One of the first steps in DREAM is *variable recovery* that recovers individual variables from the binary code. For example, functions usually allocate a space on the stack to store local variables. Expressions accessing values in this stack frame are then recovered as local variables. For efficiency, buffers are often allocated on the stack when the maximum size is known at compile time. In this case, the variable recovery step represents the buffer as local variable `v` and expressions that access items inside this buffer are represented using the address-of operator as `&v`, resulting in a decompiled code that is hard to understand. For example, reading a character from a buffer allocated on the stack is represented as `*(&v37 + dwSeed / v15 % 6)` (line 31 in Figure 1a). If a variable `v` is only accessed in the code using address expressions, i.e., `&v`, we replace these expressions by a pointer variable `v_ptr`. This creates an opportunity to further simplify pointer dereferencing expressions in which the resulting pointer variable is involved as array indexing. The previous example can be represented as `v37_ptr[dwSeed / v15 % 6]`.

IV. CODE QUERY AND TRANSFORMATION

At the core of our subsequent optimizations is our generic approach to search for code patterns and apply corresponding code transformations. The main idea behind our approach is to leverage the inference capabilities of logic programming to search for patterns in the decompiled output. To this end, we represent the decompiled code as logic facts that describe properties of the corresponding abstract syntax tree. This logic-based representation enables us to elegantly model search patterns as logic rules and efficiently perform complex queries over the code base. Usability is a key design goal, and therefore we enable users of our system to define search rules using normal C code and provide a rule compiler to compile them into the logic rules needed by our engine. We use the platform-independent, free SWI-Prolog implementation [52]. In the following, we describe our approach in detail.

A. Logic-Based Representation of DREAM IR

This step takes as input the abstract syntax tree (AST) generated by DREAM and outputs the corresponding logic facts, denoted as *code facts*. We represent each AST node as a code fact that describes its properties and nesting order in the AST. Table I shows the code facts for selected statements and expressions in DREAM's intermediate representation (IR)[3]. The predicate symbol (fact name) represents the AST node type. The first parameter is a unique identifier of the respective node. The second parameter is the unique identifier of the parent node (e.g., the containing `if` statement). Node ids and parent ids represent the hierarchical syntactic structure of decompiled code. Remaining parameters are specific to each fact and are described in detail in Table I.

We generate the code facts by traversing the input AST and producing the corresponding code fact for each visited node. The code facts are stored in a fact base $\mathcal{F}$, which will be later queried when searching for code patterns. As a simple example illustrating the concept of code facts, we consider the code sample shown in Figure 4a. The corresponding code facts for the function body are shown in Figure 4c. The body is a sequence ($id = 3$) of two statements: an `if-then-else` statement ($id = 4$) and a `return` statement ($id = 14$). These two statements have the sequence node as their parent and their order in the sequence is represented by the order of the corresponding ids inside the sequence code fact.

B. Transformation Rules

The logic-based representation of code enables us to elegantly model search patterns as *inference rules* of the form

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{C}$$

The top of the inference rule bar contains the premises $P_1, P_2, \dots, P_n$. If all premises are satisfied, then we can conclude the statement below the bar C. The premises describe the properties of the code pattern that we search for. In case

[3]We cannot show the full list due to space restrictions.

	Code Fact	Description
Statements	$\texttt{sequence}(id, p_{id}, [\#s_1, \ldots, \#s_n])$	*sequence* of statements $s_1, \ldots, s_n$
	$\texttt{loop}(id, p_{id}, \tau, \#e_c, \#s_b)$	*loop* of type $\tau \in \{\tau_{\textbf{while}}, \tau_{\textbf{dowhile}}, \tau_{\textbf{endless}}\}$ and continuation condition e_c and body s_b
	$\texttt{if}(id, p_{id}, \#e_c, \#s_{\texttt{then}}, \#s_{\texttt{else}})$	*if* statement with condition e_c, the `then` part $s_{\texttt{then}}$, and the `else` part $s_{\texttt{else}}$
	$\texttt{switch}(id, p_{id}, \#e_v, [\#s^1_{\texttt{case}}, \ldots, \#s^n_{\texttt{case}}])$	*switch* statement with variable e_v and a set of cases $s^1_{\texttt{case}}, \ldots, s^n_{\texttt{case}}$
	$\texttt{case}(id, p_{id}, \#e_{\texttt{label}}, \#s)$	*case* statement with a label $e_{\texttt{label}}$ and a statement s
	$\texttt{assignment}(id, p_{id}, \#e_{\texttt{lhs}}, \#e_{\texttt{rhs}})$	*assignment* of the form $e_{\texttt{lhs}} = e_{\texttt{rhs}}$
	$\texttt{return}(id, p_{id}, \#e)$	*return* statement that returns expression e
	$\texttt{break}(id, p_{id})$	*break* statement
Expressions	$\texttt{call}(id, p_{id}, \#e_{\texttt{callee}}, [\#e^1_{\texttt{arg}}, \ldots, \#e^n_{\texttt{arg}}])$	*call* expression of the function $e_{\texttt{callee}}$ with arguments $e^1_{\texttt{arg}}, \ldots, e^n_{\texttt{arg}}$
	$\texttt{operation}(id, p_{id}, \texttt{op}, [\#e^1_{\texttt{e}}, \ldots, \#e^n_{\texttt{e}}])$	*operation* (e.g., addition or multiplication) with operand `op` involving expressions $e^1_{\texttt{e}}, \ldots, e^n_{\texttt{e}}$
	$\texttt{ternaryOp}(id, p_{id}, \#e_c, \#s_{\texttt{then}}, \#s_{\texttt{else}})$	*ternary operation* of the form $e_c \texttt{?} s_{\texttt{then}} \texttt{:} s_{\texttt{else}}$
	$\texttt{numericConstant}(id, p_{id}, v)$	*numeric constant* of value v
	$\texttt{stringConstant}(id, p_{id}, v)$	*string constant* of value v
	$\texttt{memoryAccess}(id, p_{id}, \#e_{\texttt{address}})$	*memory access* to address $e_{\texttt{address}}$
	$\texttt{localVariable}(id, p_{id}, \texttt{name}, \tau)$	*local variable* with name `name` and type τ
	$\texttt{globalVariable}(id, p_{id}, \texttt{name}, \tau)$	*global variable* with name `name` and type τ
	$\texttt{identifier}(id, p_{id}, \#e_{\texttt{var}})$	*identifier* represents the occurrence of a variable $e_{\texttt{var}}$ in an expression p_{id}

TABLE I: Logic-based predicates for the DREAM IR. Each predicate has an id to uniquely represent the corresponding statement or expression. The second argument of each code fact is the parent id pid that represents the id of containing AST node. For a statement or expression e, we denote by $\#e$ the id of e.

```
1 int foo(int a, int b)
2 {
3   int x;
4   if(a > b)
5     x = a;
6   else
7     x = b;
8   return x + 32;
9 }
```

(a) Exemplary code

```
1 int foo(int a, int b)
2 {
3   int x = max(a, b);
4   return x + 32;
5 }
```

(b) Transformed code

```
localVariable(0, 'int', 'a').
localVariable(1, 'int', 'b').
localVariable(2, 'int', 'x').
sequence(3, _, [4, 14]).
if(4, 3, 5, 8, 11).
operation(5, 4, '>', [6, 7]).
identifier(6, 5, 0).
identifier(7, 5, 1).
assignment(8, 4, 9, 10).
identifier(9, 8, 2).
identifier(10, 8, 0).
assignment(11, 4, 12, 13).
identifier(12, 11, 2).
identifier(13, 11, 1).
return(14, 3, 15).
operation(15, 14, '+', [16, 17]).
identifier(16, 15, 2).
numericConstant(17, 15, 32).
```

(c) Code facts

Fig. 4: Code representations.

```
1 if_condition(Condition) :-
2   if(_, _, Condition, _, _).
3
4 assignment_to_local(Assignment, VarName) :-
5   assignment(Assignment, _, Lhs, _),
6   identifier(Lhs, Assignment, Variable),
7   localVariable(Variable, _, VarName).
```

Fig. 5: Sample search patterns.

of code queries, the conclusion is to simply indicate the existence of the searched pattern. For code transformation, the conclusion represents the transformed form of the identified code pattern.

We realize inference rules as Prolog rules, which enables us to ask Prolog queries about the program represented as code facts. Figure 5 shows two simple examples that illustrate the idea of modelling code search patterns as Prolog rules. The rule `if_condition` searches for condition expressions used in `if` statements. Rule parameters are Prolog variables that represent the pieces of information to be extracted from the matched pattern. The rule body represents the premises that must be fulfilled in order for the rule to return a match. At a high level, when a query is executed, Prolog tries to find a satisfying assignment to the variables of the rule that makes it consistent with the facts. For example, the query `if_condition(Condition)` executed on the fact base in Figure 4c returns the match {`Condition=5`}, the id of the code fact corresponding to the condition of the `if` statement in Figure 4a. This unification is done by matching the rule only premise with the corresponding code fact of the `if` statement.

A very powerful aspect of logic rules is that the corresponding queries can be adapted for multiple purposes. For example, the second rule `assignment_to_local` searches for assignments to a local variable given its name. Using a concrete variable name, the query returns all assignments to the corresponding variable (e.g., `assignment_to_local(Assignment, 'x')`). On the other hand, using a Prolog variable for the name, the query returns all assignments to all variables (e.g., `assignment_to_local(Assignment, Name)`).

Transformation rules can be written in normal C code. Figure 6 shows a sample transformation rule that searches for `if` statements that compute the largest of two values and replace them by a call to the `max` library function. A transformation rule consists of two parts: *rule signature* and *code transformation*. The rule signature describes the code pattern to be searched for and is written as normal C function declaration: the list of parameters, denoted as *rule parameters*, represents the variables that need to be matched to the actual variables by Prolog inference engine so that the transformed code can be constructed. The function body represents the code pattern. The transformation part describes the transformed code that should replace the matched pattern.

```
Signature:
  max(result, v1, v2){
    if(v1 > v2)
      result = v1;
    else
      result = v2;
  }
Transformation:
  result = max(v1, v2);
```

Fig. 6: Sample transformation rule.

We compile transformation rules into logic rules that can be used by Prolog's inference engine. To this end, we parse the rule body and then traverse the resulting AST. For each visited AST node, we generate the corresponding code fact. Here, we use Prolog variables for the generated fact identifiers. These variables will be then bound to the actual identifiers from the fact base when the inference engine finds a match. Finally, the compiled rule is stored in the rule base $\mathcal{R}$ and the corresponding query in the query base $\mathcal{Q}$.

C. Applying Transformation

We first initialize Prolog with the code base $\mathcal{F}$ and the rule base $\mathcal{R}$. We then iteratively apply the queries in the query set $\mathcal{Q}$. If a match is found, the inference engine *unifies* the rule arguments to the identifiers of the corresponding code facts. In this case, we construct the equivalent transformed code. To this end, we first parse the transformation string to construct the corresponding AST. During this process, we use the corresponding AST node for each rule argument to get the transformed code in terms of the original variables from the initial code base. For example, applying the sample rule in Figure 6 to the fact base shown in Figure 4c returns one match: $\{$`result` $=$ `x`, `v1` $=$ `a`, `v2` $=$ `b`$\}$. This enables us to replace the complete `if` statement by the function call `x = max(a, b)` to get the code shown in Figure 4b. Finally, we update the fact base $\mathcal{F}$ so that it remains consistent with the AST.

The code query and transformation engine is the basis for our subsequent code optimizations that identify certain code patterns and apply corresponding transformations aimed to simplify code and improve readability.

V. Control-Flow Simplification

In this section we present our techniques to simplify the control flow of decompiled code.

A. Loop Transformation

Compiler optimizations often change the structure of loops in the source code. While this optimization is aimed to increase efficiency, the resulting loop structure becomes less readable, reducing the quality of decompiled code. A well-known loop optimization is *inversion*, which changes the standard `while` loop into a `do-while` loop wrapped in an `if` conditional, reducing the number of jumps by two for cases where the loop is executed. That is, loops of the form `while(e){...}` are transformed into `if(e){do{...}while(e);}`. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a pipeline stall. Additionally, if the initial condition is known to be true at compile-time and is side-effect-free, the `if` guard can be skipped.

Here we make the observation that `while` loops are more readable than `do-while` loops since the continuation condition is clear from the start. Moreover, some `while` loops can be further simplified into `for` loops where the initialization statement, continuation condition, and the increment statements are clear from the start. Based on this observation, we analyze `do-while` loops and check if they can be transformed into `while` loops. Here, we distinguish between to cases:

Guarded `do-while` loops. loops of the form `if(c1){do{...}while(c2);}` are transformed into `while(c2){...}` if it can be proven that $c_1 == c_2$ at the start of the first iteration of the loop. Note that c_1 and c_2 does not have to be identical logical expressions. As an example, we consider the code sample shown in Figure 7a. The conditions `*(_BYTE *)v7 != 0` and `*(_BYTE *)(v8 + v7)!= 0` both yield the same Boolean value at the entry of loop. Note that the reaching definition of variable `v8` at this point is `v8 = 0`.

Unguarded `do-while` loops. For these loops we only check if the loop condition is true for the first iteration. In this case, the loop can be transformed into `while` loop.

To check the value of logic expressions at loop entry, we compute the set of definitions for loop variables that reach the loop entry. To this end, we perform a fixed-point intraprocedural *reaching definitions* analysis, a standard problem from compiler design [41, p. 218]. Often the reaching definitions for loop variables are assignments of constant values that represent the initial value of a loop counter. This makes it easy to substitute this initial value in the logic expressions and check for equivalence at loop entry.

B. Function Outlining

Function inlining is a well-known compiler optimization where all calls into certain functions are replaced with an in-place copy of the function code. This improves runtime performance since the overhead of calling and returning from a function is completely eliminated. In the context of code obfuscation, inlining is a powerful technique [19]. It makes reverse engineering harder in two ways: first, several duplicates of the same code are spread across the program. As a result, analysts end up analyzing several copies of the same code. Second, internal abstractions such as the calling relationships between functions in the program are eliminated.

Reversing function inlining is valuable for the manual analysis of malware. As a simple example illustrating the benefits of function outlining, we consider the excerpt code from the Cridex malware family shown in Figure 7. Each of the two loops in Hex-Rays decompiled code shown in Figure 7a computes the length of a string by incrementing the counter by one for each character until the terminating null-character is found. DREAM^{++} identified these two blocks as an implementation of the `strlen` library function and replaced

```
int sub_408A70(int a1, int a2){
  [...]
  v8 = 0;
  if ( *(_BYTE *)v7 )
  {
    do
      ++v8;
    while ( *(_BYTE *)(v8 + v7) );
  }
  v9 = 0;
  if ( *(_BYTE *)a1 )
  {
    do
      ++v9;
    while ( *(_BYTE *)(v9 + a1) );
  }
  if ( v8 == v9 ){
    [...]
  }
  [...]
}
```

(a) Hex-Rays

```
int sub_408A70(char * str2, void * a2){
  [...]
  len1 = strlen(str1);
  if(len1 == strlen(str2)){
    [...]
  }
  [...]
}
```

(b) DREAM^{++}

Fig. 7: Excerpt from the code of the Cridex malware family showing the code inlining technique.

them with corresponding function calls as shown in Figure 7b. This simple example gives insights into the benefits of function outlining for code analysis.

1) *Compact code.* Replacing a code block by the equivalent function call eliminates duplicate code blocks and results in a more compact decompiled output. The whole code block is replaced by a function call whose name directly reveals the functionality of the code block. Moreover, temporary variables used inside the block are removed from code, reducing the number of variables that an analyst should keep track of.
2) *Meaningful variable names.* Outlined functions have known interfaces that include the names of their parameters. These names represent their semantics and reveals important information about the variable job. We leverage this information to give meaningful names the variables in the decompiled output.
3) *Improved Type Recovery.* Approaches to recover types from binary code such as [39, 40] rely on *type sinks* as reliable starting points. Type sinks are points in the program where the type of a given variable is known. This includes calls to functions whose signatures are known. Outlining a function generates a new type sink that can be used to improve the performance of type inference algorithms.
4) *Recovering inter-dependencies.* Function outlining implicitly recovers calling relationships between the inlined function and the functions calling it. That is, it identifies points in the program that call the function. Calling relationships are very important for manual reverse engineering. After having analyzed a given function, malware analysts can draw conclusions about the calling functions.

The original DREAM decompiler contained hard-coded signatures for a minimal set of string functions, which are hard to extend. We leverage our code query and transformation engine to easily include multiple transformation rules for several functions that copy, compare, compute the length, and initialize buffers. For example, we handle `strcpy`, `strlen`, `strcmp`, `memset`. For string functions, both 8-bit and 16-bit character versions are handled. We also include signatures for the version of string functions that take buffer length as argument.

Users of our system can easily add new transformation rules to handle new functions. When an analyst observes a repeating code pattern, she can simply write a transformation rule that replaces the whole code block by a function call with a name that represents its functionality. All other copies of the same block will be outlined. Code blocks are not only duplicated as a result of function inlining. In C, *function-like macros* are pre-processor macros that accept arguments and are used like normal function calls. These macros are handled by the pre-processor, and are thus guaranteed to be inlined.

VI. SEMANTICS-AWARE NAMING

In this section we describe several readability improvements at the level of variables in the decompiled code.

A. Meaningful Names

Variable names play an important role when analyzing source code. These names reveal valuable information about the purpose of variables and how they are used in the program. We give variables meaningful names based on the context in which they occur. Here we distinguish the following cases:

Standard library calls. With well-defined API, standard library calls are important source of variable names. For example, the Windows API `URLDownloadToFile`, which downloads data from the Internet and saves them to a file, takes five arguments. Among them one argument, named `szURL`, represents the URL to download. A second argument, named `szFileName`, represents the name or full path of the file to create for the download. By analyzing library function calls and returns, we rename variables used as parameters or return values, directly revealing their purpose to the analyst.

Context-based naming. The way a variable is used in code gives insights into its purpose. We analyze the context in which variables are used to provide meaningful names to them. More specifically, we distinguish the following cases:

1) *Loop counters*. We query the decompiled code for *counting loops*, i.e., loops that update a variable inside their body and then test the same variable in their continuation condition. Counting variables in short `for` loops are renamed to `i`, `j`, or `k`. Counting variables for other loops are renamed to `counter`.

2) *Array indexes*. We rename variables used as indexes for arrays to `index`
3) *Boolean variables*. Variables that contain the result of evaluating logic expressions are renamed to `cond`. This encodes the fact that they represent testing a condition in the variable name.

When multiple variables are identified that can take the same name, we add subscripts to the default names to have unique names. For example if three loop counters are identified, they are renamed to `counter1`, `counter2`, and `counter3`.

B. Named Constants

Constants are important corner pieces in the process of reverse engineering. For example, some cryptographic algorithms uses magic numbers, and several file formats include magic numbers to identify the file type. Also standard library functions assign specific constants to special meanings. Usually these numbers have a textual representation in the source code. We use two sources to identify these special constants.

Library API constants. Many functions in the C standard library and Windows API define special named constants. These constants have a specific meaning and are thus given representative names. During compilation compilers replace this symbolic representation of the constant by the corresponding numeric value. For example, the function `CreateFile` uses the constant `GENERIC_READ` to request a read access to the opened file. This becomes `0x80000000` in the binary. To recover the symbolic, easily remembered names of these constants, we check for the occurrence of named constants for a wide range of library functions. For example, this would transform the function call `CreateFileA(f, 0x80000000, 1, ...)` into the more readable form `CreateFileA(f, GENERIC_READ, FILE_SHARE_READ, ...)`.

File magic numbers. For many file types, a file starts with a short sequence of bytes (mostly 2 to 4 bytes long) to uniquely identify its type. Detecting such constants in files is a simple and effective way of distinguishing between many file formats. For example, DOS MZ executable file format and its descendants (including NE and PE) begin with the two bytes `4D 5A` (characters `'MZ'`). Malware usually downloads files from its C&C server at runtime and may check which file type it received. We check if such constants are used in the conditions of flow control statements.

VII. User Study Design

The goal of our study is to test the readability of the code produced by our improved decompiler DREAM^{++} compared to the academic predecessor DREAM and the industry standard Hex-Rays[4]. We planned a user study in which participants have to solve a number of reverse engineering tasks with different decompilers. The participants' task was to analyze the code snippets and answer a number of questions about the functionality of the code. Our web-based study platform showed the code in split screen together with the questions. Participants could edit the code to help with the analysis.

To measure user perception, after each task we asked the participants for feedback and a couple of questions regarding readability. Each participant got a number of code snippets produced by different decompilers without being told which decompiler was being used, so we would get an unbiased evaluation of the code. Only at the end of the study we showed the users the code produced by all decompilers side by side and asked them to give an overall rating for the decompilers.

A. Task Selection

To minimize the risk of bias, i.e., subconsciously selecting tasks which would favor our decompiler, we approached three independent professional malware analysts for the process of task selection. The analysts were known to us, but were not involved in the study or the work on the decompiler. We told them that we wanted to conduct a study on malware analysis and requested that they supply us with malware code snippets they themselves had to analyze in the course of their work. We requested that the snippets fulfil some sort of function that should be understandable without needing the rest of the malware code. In total we got 8 snippets of code. Two of the snippets contained an XOR based encryption/decryption algorithm, so we removed one of those and were left with 7 malware snippets. We ran a pre-study which will be described in more detail below to test the tasks. For this pre-study, an even number of tasks was preferable so we added one additional code snippet. Based on the results of the pre-study we removed this and one other snippet, so we are left with 6 snippets. We grouped these snippets into two groups: *Medium* (three tasks), and *Hard* (three tasks). In the following, we describe the tasks in detail.[5]

1) Encryption: Encoding functions are used widely in malware as well as benign applications. Malware can encrypt exchanged messages with C&C servers and encode internal strings to avoid static analysis. This task is a function from the Stuxnet malware that decrypts the `.stub` section which contains Stuxnet's main DLL.

2) Custom Encoding: This task is an XOR encryption/decryption function from the Stuxnet malware family. This function performs a word-wise XOR with `0xAE12` and is used by many Stuxnet components to disguise some strings.

3) Resolving API Dynamically: In order to avoid static analysis, malware usually avoids listing the API functions it needs in the import table. Instead, it can resolve them dynamically at runtime. The task is a function from the Cridex malware that takes as input the name of an API function and returns the corresponding starting address.

4) String Parsing: Malware often receives commands and configuration files from a C&C server. Thus, it needs to parse these commands to extract parameters and other information from C&C messages. This task is the injects parsing function from the URLZone malware. The function examines a string

[4]We opted to not compare DREAM^{++} to any other decompilers because of the upper bound of the number of participants (see §VII-C).

[5]For information on the snippets which were removed the reader is referred to the supplemental document.

for the first occurrence of the sequence `%[A-Z0-9]%` and returns a pointer to the start of such a string and its length.

5) Download and Execute: A very common function is to download an executable from a C&C server and later execute it. This can for example be the case for pay-per-install services [11]. The task involves analyzing the update mechanism of the Andromeda malware. The snippet downloads a file from a remote server and checks if it is a valid PE executable or a Zip archive containing an executable. In this case the file is saved on disk and executed.

6) Domain Generation Algorithm: Malware is often equipped with domain generation algorithms (DGA) to dynamically generate domain names used for C&C (e.g., depending on seed values such as the current date/time and Twitter trends) [2]. It is a powerful technique to make botnets more resilient to attacks and takedown attempts. The task contains the DGA of the Simda malware.

Due to space constraints, we cannot present the full decompiled malware source code used in our study in this paper. For this reason, we have created a supplemental document which can be accessed under the following URL: https://net.cs.uni-bonn.de/fileadmin/ag/martini/Staff/yakdan/sm-oakland-2016.tar.gz. Here, we present all data related to our user study including the decompiled code from the three decompilers as well as the set of tasks.

B. Pre-Study

Before conducting our main user study, we conducted a small scale pre-study for the following purpose: it is best practice to test user studies in a pre-study to unearth problems with the task design and the study platform. We also wanted to check whether our cognitive walkthrough and informal interviews had missed any important issues that should be dealt with before conducting the full study. To plan the study and the appropriate compensation, we needed estimates on how long each task would take.

Since malware analysis is a highly complex task requiring specialized skills, we recruited students who had successfully completed our malware boot camp. The malware boot camp is a lab course held each semester at our university, in which students are introduced to the field of malware and binary code analysis. For the pre-study we recruited two of these students.

We conducted the pre-study in our usability lab and used a think-aloud protocol, asking participants to vocalize their thought processes and feelings as they performed their tasks. We chose this protocol to obtain insights into the users' thought processes, and the barriers and problems they faced. In the pre-study we only tested DREAM^{++} and Hex-Rays. The first participant got four tasks decompiled with DREAM^{++} and four decompiled with Hex-Rays and had to answer questions about the functionality of the code. The second participant got the inverse selection. Assignment was randomized. After each task, the participants were asked to provide feedback about the quality of the code they analyzed. Then, they were shown the output generated by the other decompiler and asked which code they find more readable and how long they think they would take to analyze that output.

1) Pre-Study Results: There were only minor bugs with the study platform and the participants understood the task descriptions without problems. Based on their think-aloud feedback, we did not find any open problems that had not been discovered during the cognitive walkthrough. We also ranked our tasks based on reported difficulty.

In the main study, we wanted to additionally test DREAM implying a smaller number of task samples since we never show any participant code from different decompilers for the same task. Based on the pre-study we estimated that the main study should be completed in 3 hours. For the detailed pre-study results including the participants feedback, we refer to Appendix E.

C. Methodology

We conducted a within-subjects design experiment where participants had to analyze the code snippets decompiled by the three different decompilers (DREAM^{++}, DREAM and Hex-Rays). To begin with, we provided a detailed explanation of the concept and the procedure to participants. Participants were allowed to use the Internet during the study since this mirrors how analysts actually work. The goal was to remove all aspects not related to the quality of decompiled code. The tutorial was followed by a training phase to ensure that the participants were familiar enough with the system to avoid system-related mistakes. We provided a sample code snippet and instructed the participants to rename variables in it and look for information about a library function online.

The main user study is almost unchanged from the pre-study. The methodology differs in the following points. The decompiler names were blinded so as not to bias the participants. Also, the study was not conducted in the lab but via our online study platform. This decision was made for several reasons: First, not all of the students who had completed the malware boot camp were still living locally and we wanted to maximise our recruiting pool, since participants at this level are very scarce. Second, we also wanted to conduct the study with professional malware analysts and it is unrealistic to expect them to come to the lab. We decided to conduct the entire study online in order to keep the results comparable.

1) Variables and Conditions: In our experiment, we have two independent variables:

Decompiler: Decompiler used to solve a given task and has three conditions: DREAM^{++}, DREAM, and Hex-Rays. Hex-Rays is the leading industry decompiler that is widely used by malware analysts. Therefore, we compare DREAM^{++} to Hex-Rays to examine whether our approach improves the current state of malware analysis. We tested the latest Hex-Rays version, which is 2.2.0.150413 as of this writing. Also, we compare DREAM^{++} to the original DREAM decompiler to evaluate the usefulness of our new extensions.

Difficulty: A within-subjects factor that represents the difficulty of the task. Based on the results from our pre-study (§VII-B), we grouped the tasks according to their

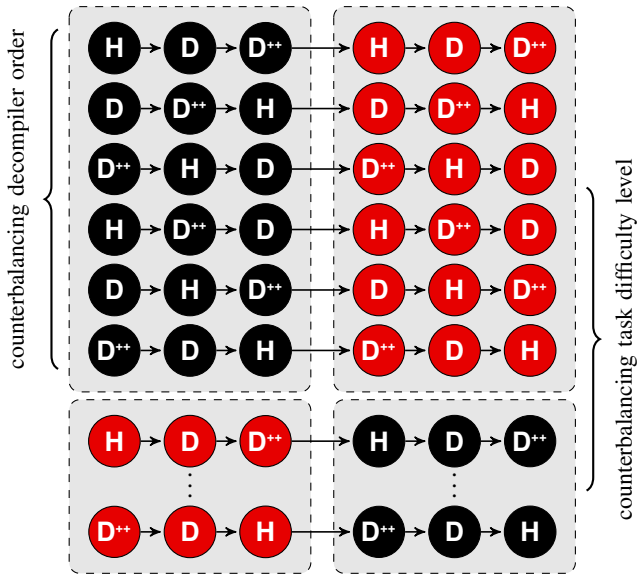


Fig. 8: Counterbalancing the order of decompiler and difficulty levels. Nodes in each horizontal sequence represent the tasks performed by one participant. Letters denote the used decompiler for the task and colors represent task difficulty level: medium (black) or hard (red).

difficulty into two groups (*medium* and *hard*), each containing three tasks.

2) Condition Assignment: We chose a within-subjects design since personal skill is a strong factor in performance. To avoid learning and fatigue effects in our within-subjects design, the order in which participants used decompilers within each difficulty level, and the difficulty levels were permutated using the counterbalanced measures design. Figure 8 shows the details of our counterbalance design: Within each difficulty level, there are 6 possibilities to order the three decompilers. The two difficulty levels are also permutated (red vs. black in the figure). Counterbalancing the order of difficulty level doubles the total numbers of possible orderings. We opted to balance on difficulty level instead of task level since this gives us a counterbalance permutations of 12 $(3! * 2!)$ instead of 4320 $(3! * 6!)$. Since we could not hope to recruit 4320 participants we opted for the compromise of recruiting multiples of 12 participants using all rows of our counter. This design ensures that each decompiler and each difficulty level gets the same exposure across the study and minimizes the overall learning and fatigue effects. This also guarantees that each participant gets the same number of medium and hard tasks for each decompiler. This is important to control for individual differences between participants and avoid skewing the results by eliminating the possibility of a skilled and motivated participant performing all of her tasks using one decompiler, while a less skilled participant performs her tasks with another decompiler.

3) User Perception: After finishing each task, participants are shown a brief questionnaire, where they can score the quality of the code produced by the decompiler, and a text field for additional feedback. Here, the participants are able to see the code again. We asked a total of 8 questions, 6 on readability properties and 2 on trust issues. Similar to the System Usability Score (SUS) [8], the questions are counterbalanced (positive/negative) to minimize the response bias, e.g., *"This code was easily readable"* and *"It was strenuous to understand what this code did"*. The full question set can be found in Appendix B.

In addition to the questionnaire after each task, at the end of the study, we asked the participants about an overall rating on a scale from 1 (worst) to 10 (best). During this step they were shown the code snippets for every task and every decompiler to facilitate the direct comparison. To avoid biasing the participants the decompilers were named M, P and R.

4) Statistical Testing: For all statistical hypothesis testing, we opted for the common significance level of $\alpha = 0.05$. To account for multiple testing, all p-values are reported in the Holm-Bonferroni corrected version [32].

Continuous tests such as time intervals or user-ratings are tested with a Holm-Bonferroni corrected Mann–Whitney U test (two-tailed). Rather than testing all pairs for the pairwise comparison, we only perform tests with our decompiler (DREAM^{++} vs. DREAM and DREAM^{++} vs. Hex-Rays). The effect size is reported by mean comparisons and the usage of the *common language effect size* method. Categorical contingency comparisons are tested with the two-tailed Holm-Bonferroni corrected Barnard's Exact test.

VIII. User Study

In this section, we present our study results: after discussing the demographics of our participants, we proceed with the code analysis experiment, and then the user perception discussion.

A. Participants

We sent 36 invitations to students who had completed the malware boot camp at our university with the aim of getting 24 participants to join for a compensation of 40 Euro. The malware boot camp is a lab course held each semesters at our university, in which students are introduced to the field of malware and binary code analysis. The invitation text can be found in Appendix A.

22 students took part in the study. One student began the study but only completed one task, so we removed this student from the sample, leaving us with 21 participants. Among the student group, the median age was 26 years. The oldest participant was 31 years old and the youngest was 19 years old. Two participants did not report their age. One of the participants was female. The median malware analysis experience in years is 1 year. One participant reported to have 14 years malware analysis experience, 7 participants reported less than a year.

We also invited 31 malware experts from commercial security companies. Based on informal talks, we were told that offering these experts the same compensation would

Decompiler	Avg. Score	p-value	Pass	Fail	p-value
Students					
DREAM^{++}	70.24		30	12	
DREAM	50.83	**0.002**	16	26	**0.002**
Hex-Rays	37.86	**<0.001**	11	31	**<0.001**
Experts					
DREAM^{++}	84.72		15	3	
DREAM	79.17	0.234	15	3	0.570
Hex-Rays	61.39	0.086	9	9	0.076

TABLE II: Aggregated experiment results.

not motivate them since time is more valuable than money. However, it was suggested that many would be intrinsically motivated to help because any improvements made in this domain would ultimately benefit them. Based on this feedback, we opted to offer early access to *an academic decompiler* for the participants and give them access to DREAM^{++} after the study. 17 malware analysts started the study. However, 8 looked at the first task only without actually starting the study. In total 9 malware analysts took part in the study, all male with a median age of 30 years (2 did not disclose their age and gender).

B. Malware Analysis Experiment

We assigned a weight to each question according to its importance in understanding the task and scored all answers. We conducted a two-pass scoring approach for calibration purposes. Since the answers contained variable names and referred to loop structures, it was not possible to blind this part of the evaluation. Table II summarizes the results of the code analysis experiment. Unsurprisingly, professional analysts performed better than students. In both groups participants performed better when using DREAM^{++} compared to both DREAM and Hex-Rays. In the student group, our sample size provides sufficient statistical power to confirm a statistically significant difference in scores. Another interesting observation is that experts did much better with Hex-Rays than the students did, suggesting that they have gotten used to working with code produced by Hex-Rays. A more detailed table is provided in Appendix C.

To get a better feeling for the results, we additionally marked each task as a pass if participants scored over 70%. This level was chosen based on our judgement that these tasks had been answered sufficiently, meaning that their results would be useful in a malware analysis team. Here, the difference between the professional analysts and the students becomes more apparent. However, in both groups DREAM^{++} performed better than the competition.

We also measured the time needed to complete the different tasks. We cannot do a statistical analysis of the time means because of a limited number of samples. Note that only successfully completed tasks can be considered in that comparison. Unfortunately, many participants failed or gave up tasks using Hex-Rays and DREAM. Nonetheless, there is a trend for participants solving tasks faster with DREAM^{++}, but more samples are needed to quantify this reliably. A detailed overview on task level, including average time spent on a task, can be found in Appendix C.

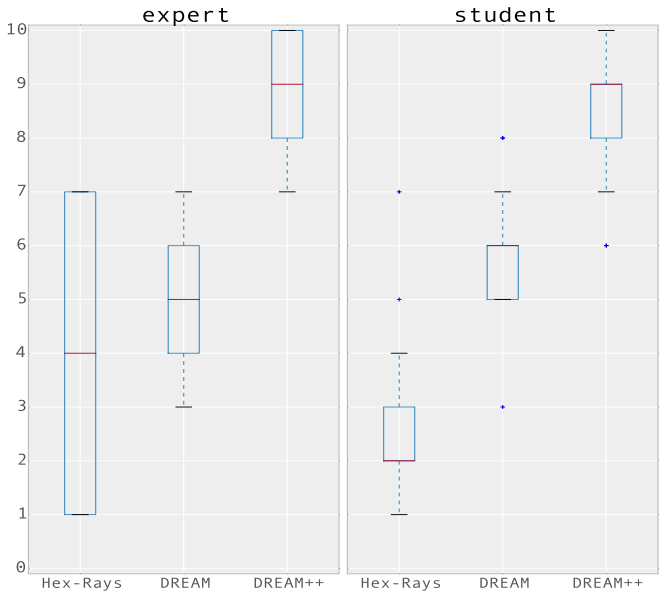


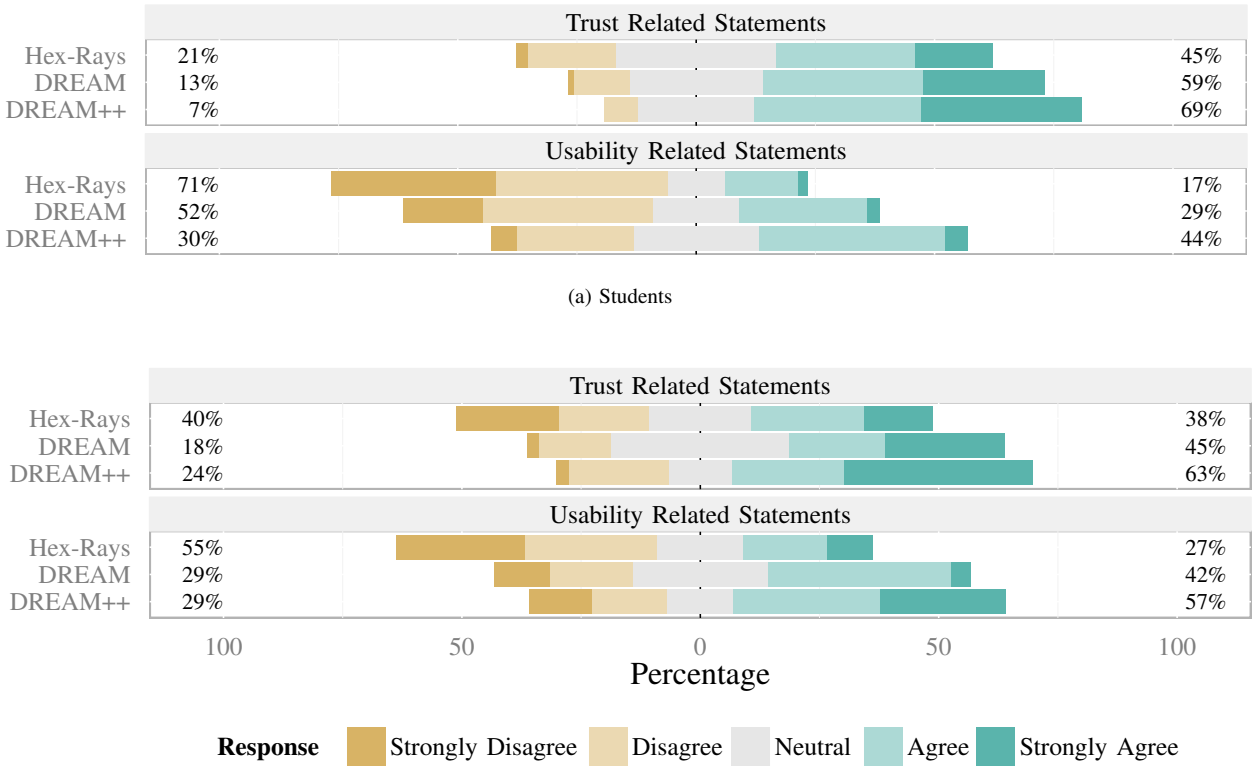
Fig. 10: Boxplot for decompiler rating.

C. User Perception

To measure user perception, after each task, we asked our participants 8 questions. The full list of questions can be seen in Appendix III. Figure 9 summarizes the aggregated user perception results. Here, we distinguish between two groups of questions: usability perception (questions 1-6) and trust perception (questions 7&8). The trust issue is interesting because decompilers do make mistakes or create misleading code and it is common for malware analysts to fall back to analyzing assembly code. Thus, we wanted to see whether there were different levels of trust in the code.

A pairwise comparison between the decompilers shows a high statistical significance difference ($p < 0.001$) for usability related questions for experts and students. The trust related questions showed only a statistically significant difference in comparison to Hex-Rays ($p < 0.001$ in the student group, $p = 0.03$ in the expert group). The color coding in Figure 9 shows the agreement level and the percentage numbers on the right and left hand side summarize the percentage of participants who rated positively and negatively respectively.

After finishing all tasks, we asked for an overall rating for each decompiler. Figure 10 shows a boxplot with a rating distribution overview. It can be clearly seen that DREAM^{++} achieves higher scores than the competition for both students and experts (adjusted $p < 0.001$ for all pairwise comparisons). Interestingly, while it can be clearly seen that the experts rated Hex-Rays better compared to the students, they were also more enthusiastic about DREAM^{++}. These are very promising results, both groups clearly prefer the code produced by our improved decompiler and even though the experts cope well with Hex-Rays they gave DREAM^{++} outstanding marks.



(a) Students

(b) Experts

Fig. 9: Aggregated participant agreement with the statements related to usability perception (6 questions) and trust in correctness (2 questions).

IX. Related Work

A wealth of research has been conducted on decompilation and the development of principled methods for recovering high-level abstractions from binary code. At a high level, there are four lines of research relevant to the work presented in this paper. First, approaches to extract binary code from executables. Second, research on recovering abstractions required for source code reconstruction. Third, work on end-to-end decompilers. Finally, techniques to query code bases and apply transformations.

Binary code extraction. A fundamental step for decompilation is the correct extraction of binary code. Kruegel et al. [37] presented a method to disassemble x86 obfuscated code. Kinder et al. [35] proposed a method that interleaves multiple disassembly rounds with data-flow analysis to achieve accurate and complete CFG extraction. The binary analysis platform BitBlaze [49] and its successor BAP [9] use value set analysis (VSA) [3] to resolve indirect jumps. Run-time packers are often used by malware-writers to obfuscate their code and hinder static analysis [51]. To handle these cases, the research community proposed approaches that rely on dynamic analysis to cope with heavily obfuscated. This include approaches to extract a complete CFG [42], extract binary code from obfuscated binaries [58], deobfuscate obfuscated executable code [20, 56]. A closely related topic is the identification of functions in binary code. Recently, security research started to explore approaches based on machine learning to solve this problem. ByteWeight [4] learn signatures for function starts using a weighted prefix tree, and recognize function starts by matching binary fragments with the signatures. Shin et al. [47] use neural networks.

Abstractions recovery from binary code. Source code reconstruction requires the recovery of two types of abstractions: data type abstractions and control flow abstractions. Previous work addressed principled methods to recover these abstractions from binary code. Recent work proposed static and dynamic approaches to recover both scalar types (e.g., integers or shorts) and aggregate types (e.g., arrays and structs). Prominent examples include REWARDS [40], Howard [48], TIE [39], and MemPick [31]. Other work [22, 27, 28, 34] focused on C++ specific issues, such as recovering C++ objects, reconstructing class hierarchy, and resolving indirect calls resulting from virtual inheritance.

Early work on control structure recovery relied on interval analysis [1, 18], which deconstructs the CFG into nested regions called intervals. Sharir [46] subsequently refined interval analysis into structural analysis. Structural analysis recovers the high-level control structure by matching regions in the CFG against a predefined set of patterns or region schemas. Engel et al. [25] extended structural analysis to handle C-specific control statements. They proposed a Single Entry Single Successor (SESS) analysis as an extension to structural analysis to handle the case of statements that exist before `break` and `continue` statements in the loop body.

Significant advances has been made recently in the field of control flow structure recovery. Schwartz et el. [45] proposed two enhancements to vanilla structural analysis: first, *iterative refinement* chooses an edge and represents it using a `goto` statement when the algorithm cannot make further progress. This allows the algorithm to find more structure. Second, *semantics-preserving* ensures correct control structure recovery. Yakdan et. al. [57] proposed *pattern-independent* control flow structuring, an approach that relies on the semantics of high-level control constructs rather than the shape of the corresponding flow graphs. Their method is a departure from the traditional pattern-matching approach of structural analysis and is able to produce a `goto`-free output.

Code query and transformation. Several code query technologies based on first-order predicate logic have been proposed. They are mainly used in software engineering for detecting design patterns or patterns of problematic design. These techniques support specific source languages and they either introduce new languages for modeling code queries such as CrocoPat [5, 6] and SOUL [54], or users of these tools have to write logic rules directly such as JTransformer [36]. Our code query and transformation engine is based on the DREAM IR and enables malware analysts to directly write transformation rules as normal C code.

Decompilers. Cifuentes laid the foundations for modern decompilers. In her PhD thesis [16], she presented several techniques for decompiling binary code that spans a wide range of techniques from data-flow analysis and control-flow analysis. These techniques were implemented in dcc, a decompiler for Intel 80286/DOS to C. Cifuentes et al. also developed *asm2c*, a SPARC assembly to C decompiler, and used it to decompile the integer SEPC95 programs [17].

Van Emmerik proposed to use the Static Single Assignment (SSA) form for decompilation in his PhD thesis [24]. His work shows that SSA enables efficient implementation of many decompiler components such as expression propagation, dead code elimination, and type analysis. His techniques were implemented in the open-source Boomerang decompiler. Although faster than interval analysis, it recovers less structure. Another open-source decompiler [14] is based on the work of van Emmerik.

Chang et el. [15] created a modular framework for building pipelines of cooperating decompilers. Decompilation is performed by a series of decompilers connected by intermediate languages. Their work demonstrates the possibility of using source-level tools on the decompiled source to find bugs that were known to exist in the original C code.

Hex-Rays is the *de facto* industry standard decompiler [30]. Hex-Rays is developed by Ilfak Guilfanov and built as plugin for the Interactive Disassembler Pro (IDA). Since it is closed source, little is known about the exact approach used. It uses an enhanced version of vanilla structural analysis and has an engine to recognize several inlined functions. There are also other decompilers available online such as DISC [38] and REC [43]. However, our experience suggests that all previously mentioned decompilers are not as advanced as Hex-Rays.

Phoenix is an advanced academic decompiler created by Schwartz et al. [45]. It is built on top of the Binary Analysis Platform (BAP) [9], which lifts sequential x86 assembly instructions into the BIL intermediate language. It also uses TIE [39] to recover types from binary code. Phoenix uses an enhanced structural analysis algorithm that can correctly recover more structure than vanilla structural analysis. Schwartz et al. were the first to measure correctness of decompiler as a whole. Their methods rely on checking if the decompiled code can pass the automatic checks written for source code.

DREAM is the newest academic decompiler developed by Yakdan et al. [57]. DREAM uses IDA to extract binary code and build CFGs. Type recovery is based on TIE. It uses a novel control-flow structuring algorithm to recover high-level control construct without relying on patterns. DREAM is the first decompiler to produce a `goto`-free decompiled output. It is assumed that this makes DREAM the decompiler which currently produces the most readable code. For this reason, we use DREAM as the basis for our work.

All presented works presented above share two common characteristics. First, they do not leverage the recovered abstraction to simplify the decompiled code, and thus they miss opportunities to improve readability. At best, minimal readability enhancements are implemented.

User studies. To the best of our knowledge we are the first to conduct user studies in the domain of malware analysis.

X. CONCLUSION

In this paper we made two contributions. First, we created a host of novel readability-focused code transformations to improve the quality of decompiled code for malware analysis. Our transformations simplify both program expressions and control flow. They also assign meaningful names to variables and constants based on the context in which they are used. Second, we validated our improvements with the first user study in this domain, involving both students and professional malware analysts. The results clearly show that our human-focused approach to decompilation offers significant improvements, with DREAM^{++} outperforming both DREAM and Hex-Rays. Despite these large improvements, we believe we have only barely scratched the surface of what can be done in this highly technical domain. We hope that our user study can serve as a template for similar studies in malware analysis.

ACKNOWLEDGEMENTS

We are grateful to Daniel Plohmann for reaching out to malware analysts and inviting them to the study. We sincerely thank all the students and experts that agreed to support our research by taking part in our user study. We also thank Houssam Abdoullah for the fruitful discussions on logic-based code representation and transformation. Finally, we thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] F. E. Allen. Control Flow Analysis. In *Proceedings of ACM Symposium on Compiler Optimization*, 1970.

[2] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon. From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[3] G. Balakrishnan. *WYSINWYX What You See Is Not What You eXecute*. PhD thesis, University of Wisconsin at Madison, 2007.

[4] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[5] D. Beyer. Relational Programming with CrocoPat. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[6] D. Beyer, A. Noack, and C. Lewerentz. Efficient Relational Calculation for Software Analysis. *IEEE Transactions on Software Engineering (TSE)*, 31(2), 2005.

[7] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry. 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)*, 2015.

[8] J. Brooke. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, 2011.

[10] R. P. L. Buse and W. R. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010.

[11] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[12] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[13] J. Calvet, J. M. Fernandez, and J.-Y. Marion. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[14] F. Chagnon. Decompiler. https://github.com/EiNSTeiN-/decompiler. Page checked 8/20/2015.

[15] B.-Y. E. Chang, M. Harren, and G. C. Necula. Analysis of Low-level Code Using Cooperating Decompilers. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*, 2006.

[16] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[17] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to High-Level Language Translation. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1998.

[18] J. Cocke. Global Common Subexpression Elimination. In *Proceedings of the ACM Symposium on Compiler Optimization*, 1970.

[19] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Sciences, The University of Auckland, 1997.

[20] K. Coogan, G. Lu, and S. Debray. Deobfuscation of Virtualization-obfuscated Software: A Semantics-based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[21] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[22] D. Dewey and J. T. Giffin. Static detection of C++ vtable escape vulnerabilities in binary code. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

[23] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1976.

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

[25] F. Engel, R. Leupers, G. Ascheid, M. Ferger, and M. Beemster. 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)*, 2011.

[26] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. Symantec Corporation, 2011.

[27] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. SmartDec: Approaching C++ Decompilation. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE)*, 2011.

[28] A. Fokin, K. Troshina, and A. Chernov. Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[29] G Data SecurityLabs. Uroburos Highly complex espionage software with Russian roots. G Data Software AG, 2014.

[30] I. Guilfanov. Decompilers and Beyond. In *Black Hat, USA*, 2008.

[31] I. Haller, A. Slowinska, and H. Bos. MemPick: High-Level Data Structure Detection in C/C++ Binaries. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, 2013.

[32] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[33] The IDA Pro disassembler and debuger. http://www.hex-rays.com/idapro/.

[34] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. Recovering C++ Objects From Binaries Using Inter-Procedural Data-Flow Analysis. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW)*, 2014.

[35] J. Kinder and H. Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, 2008.

[36] G. Kniesel, J. Hannemann, and T. Rho. A Comparison of Logic-based Infrastructures for Concern Detection and Extraction. In *Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution (LATE)*, 2007.

[37] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium*, 2004.

[38] S. Kumar. DISC: Decompiler for TurboC. http://www.debugmode.com/dcompile/disc.htm. Page checked 8/20/2015.

[39] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*, 2011.

[40] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[41] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[42] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[43] REC Studio 4 - Reverse Engineering Compiler. http://www.backerstreet.com/rec/rec.htm. Page checked 8/20/2015.

[44] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[45] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[46] M. Sharir. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, 5(3-4):141–153, Jan. 1980.

[47] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[48] A. Slowinska, T. Stancescu, and H. Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[49] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, 2008.

[50] Symantec Security Response. Regin: Top-tier espionage tool enables stealthy surveillance, 2014.

[51] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. [SoK] Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers (S&P). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.
[52] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.
[53] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic Network Protocol Analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.
[54] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 2001.
[55] Z. Xu, J. Zhang, G. Gu, and Z. Lin. GoldenEye: Efficiently and Effectively Unveiling Malware's Targeted Environment. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.
[56] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.
[57] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.
[58] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation Resilient Binary Code Reuse Through Trace-oriented Programming. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

APPENDIX A
RECRUITMENT ADVERTISEMENT

To recruit our student participants, we sent the following email:

Subject: **Invitation to a Decompiler Study**

we would like to invite you to participate in a research study conducted by researchers at the University of Bonn and Fraunhofer FKIE on the quality of binary code decompilers for malware analysis. The study evaluates three state-of-the-art industrial and academic decompilers. You will be asked to complete six reverse engineering tasks. For each task you will get the decompiled code of one function from a malware sample and a set of questions regarding its functionality. That is, you will be analyzing high-level C code. The study is not aimed at testing your ability but the quality of the decompilers.

The study will take you approximately 3 hours and will take place at the University of Bonn. There will be several dates available to take part in the study. This study is anonymous, so no personally identifying information will be collected during the study and we will only report the aggregated results in a scientific publication. In appreciation of your choice to participate in the project, you will be paid 40 Euro.

To recruit our expert participants, we sent the following email:

Subject: **Invitation to a Decompiler Study**

we would like to invite you to participate in a research study conducted by researchers at the University of Bonn and Fraunhofer FKIE on the quality of binary code decompilers for malware analysis. The study evaluates three state-of-the-art industrial and academic decompilers. You will be asked to complete six reverse engineering tasks. For each task you will get the decompiled code of one function from a malware sample and a set of questions regarding its functionality. The study is not aimed at testing your ability but the quality of the decompilers.

The study will take you approximately 2 hours. You will be given the URL to our online study platform and can perform the study remotely. You can take breaks between the tasks, however the tasks themselves need to be completed uninterrupted. This study is anonymous, so no personally identifying information will be collected and we will only report the aggregated results in a scientific publication. In appreciation of your choice to participate in the project, we will pass along your comments to the developers of the decompilers for consideration. Also, you will get free access to an improved decompiler as soon as the decompiler is released. You will also be helping the malware analysis community.

APPENDIX B
SURVEY QUESTIONS

A. Questions After each Task

Tabel III shows the questions asked to the participants after finishing each task. The order of questions in this table is the same order they were presented to the participants.

B. Questions About Participants' Demographics

1. Gender?
- ○ Male
- ○ Female
- ○ Prefer not to answer

2. What is your age? (text field)

3. Employment Status: Are you currently?
- ○ Student
- ○ Other: (text field)

4. How many years experience do you have in malware analysis? (text field)

5. How many years experience do you have in reverse engineering? (text field)

6. Which binary code decompilers did you use before?
- ☐ Boomerang
- ☐ Hex-Rays
- ☐ REC
- ☐ DISC
- ☐ Other: (text field)

APPENDIX C
DETAILED STUDY RESULTS

Table IV shows the detailed study results for the student participants. For each task, the average score achieved and corresponding standard deviation are shown. Next, we show average time needed to complete the task and the corresponding standard deviation. Finally, the number of tasks that were solved successfully/unsuccessfully is mentioned.

APPENDIX D
DETAILED USER PERCEPTION

Figure 11 shows the participant agreement with each of the statements in Table III.

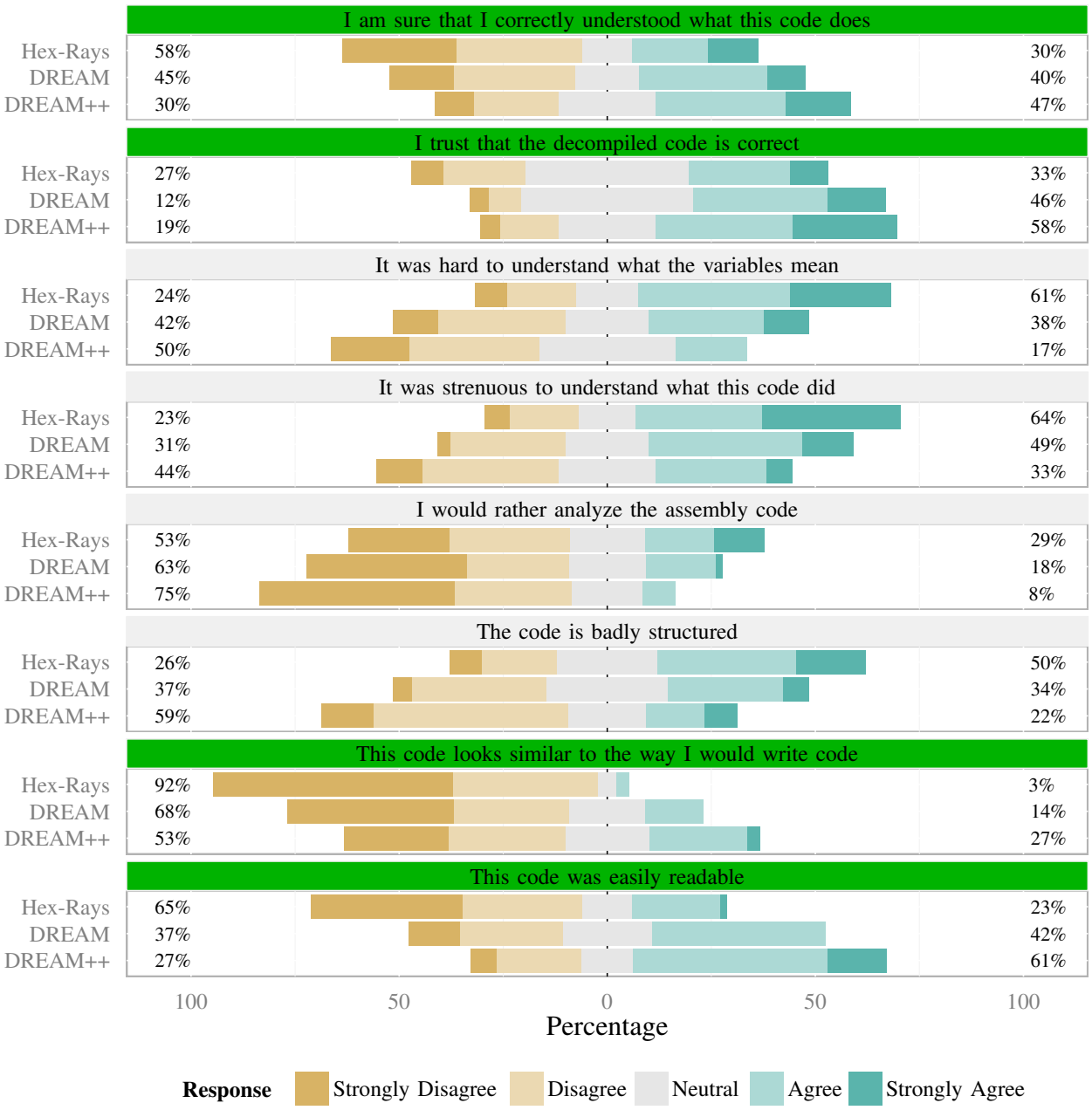


Fig. 11: Participant agreement with the statements from Table III. The positive statements are marked in green and the negative statements are marked in gray.

Statement	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
This code was easily readable	○	○	○	○	○
It was strenuous to understand what this code did	○	○	○	○	○
This code looks similar to the way I would write code	○	○	○	○	○
The code is badly structured	○	○	○	○	○
I am sure that I correctly understood what this code does	○	○	○	○	○
It was hard to understand what the variables mean	○	○	○	○	○
I trust that the decompiled code is correct	○	○	○	○	○
I would rather analyze the assembly code	○	○	○	○	○

TABLE III: Questions after each task.

	Score		Time		Result	
Decompiler	**Mean**	**Stdev**	**Median**	**Stdev**	**Pass**	**Fail**
Task 1						
DREAM++	45.71%	32.78	8.06	3.01	3	4
DREAM	56.43%	31.02	28.55	10.69	4	3
Hex-Rays	25.71%	36.3	57.42	23.65	2	5
Task 2						
DREAM++	71.88%	28.72	29.88	17.55	6	2
DREAM	61.67%	30.51	29.72	6.68	3	3
Hex-Rays	29.29%	33.85	18.58	0.0	1	6
Task 3						
DREAM++	80.83%	15.92	13.96	4.43	5	1
DREAM	32.5%	26.81	31.6	0.0	1	7
Hex-Rays	30.71%	26.65	26.94	0.0	1	6
Task 4						
DREAM++	74.29%	23.06	31.04	7.31	5	2
DREAM	55.71%	28.71	22.52	3.17	2	5
Hex-Rays	43.57%	27.87	50.34	10.86	2	5
Task 5						
DREAM++	81.25%	11.92	22.85	9.67	7	1
DREAM	80.0%	18.48	27.12	6.32	5	1
Hex-Rays	68.57%	21.99	37.13	15.65	4	3
Task 6						
DREAM++	66.67%	23.92	28.97	5.81	4	2
DREAM	30.0%	20.62	53.63	0.0	1	7
Hex-Rays	29.29%	27.31	44.12	0.0	1	6
Decompiler	**Mean**	**Stdev**	**Median**	**Stdev**	**Pass**	**Fail**

TABLE IV: Detailed study results for the students group.

Appendix E
Pre-Study Results

Table V shows an overview of the results and the comments made by the participants. Task 8 is the task we added to balance the study.

PARTICIPANT	DECOMPILER	PERFORMANCE	DURATION	PARTICIPANT FEEDBACK
Task 1: Encryption				
P1	DREAM^{++}	●	12 m	1.1 DREAM^{++}'s output is similar to what I would write 3.2 I would need 2x more time for Hex-Rays.
P2	Hex-Rays	●	16 m	1.3 DREAM^{++}'s is shorter and easier to understand.
Task 2: Custom Encoding				
P2	DREAM^{++}	●	9 m	2.1 It was easy to follow and understand the code. 2.2 Hex-Rays code is complex and I would take longer to understand.
P1	Hex-Rays	●	42 m	2.3 I was confused about the loop condition. 2.4 The code is difficult to understand. 2.5 default variable names makes it more difficult 2.6 DREAM^{++}: shorter, less variables, loop is easier to understand 2.7 I would give DREAM^{++} 8/10 and Hex-Rays 4/10
Task 3: Resolving API Dynamically				
P2	DREAM^{++}	●	16 m	3.1 DREAM^{++}'s output could be further simplified. 3.2 For the Hex-Rays output I would need at least 45 minutes. 3.3 I find the meaningful names assigned by DREAM^{++} helpful.
P1	Hex-Rays	●	23 min	3.4 Hex-Rays has several redundant variables. DREAM^{++} output has less variable. 3.5 I find the code in the last loop a spaghetti code.
Task 4: String Parsing				
P1	DREAM^{++}	●	22 m	4.1 I find the code easy to understand because it looks like the code I would write when programming. 4.2 DREAM^{++}'s output is much better. 4.3 It helped me that the DREAM^{++} has less variables.
P2	Hex-Rays	○	43 m	4.4 DREAM^{++}'s code has less variables and thus easier to understand. 4.5 The control flow is easier to understand since no `goto` statements. 4.6 It is much easier to follow the control flow in DREAM^{++} output.
Task 5: Download and Execute				
P2	DREAM^{++}	●	16 m	5.1 Named constants help. 5.2 No `goto` spaghetti code.
P1	Hex-Rays	●	36 m	5.3 `goto` statements are confusing: jumping out of the loop and then back in it. 5.4 DREAM^{++}'s output is easier to understand. One can simply read the code sequentially without worrying about these jumps. 5.5 I cannot say how much this will influence the time I need to solve the task when analyzing DREAM^{++}'s output.
Task 6: Domain Generation Algorithm				
P1	DREAM^{++}	●	33 m	6.1 The control flow inside the function is easy to understand 6.2 For the Hex-Rays code, I would need at least 60 minutes (probably 90 minutes). Maybe I would give up after that.
P2	Hex-Rays	◐	36 m	6.3 Code looks very weird. 6.4 I gave up because I do not think I could understand the code in the loop.
Task 7: Checking OS Version				
P1	Hex-Rays	●	3 m	No Comments
P2	DREAM^{++}	●	7 m	No Comments
Task 8: Persistence				
P1	DREAM^{++}	●	2 m	No Comments
P2	Hex-Rays	●	2 m	No Comments

TABLE V: Pre-study results. The third column denotes the result of performing the task: ● task is completely solved, ◐ = task is partially solved, and ○ = task is not solved.