Ghidra Decompiler

Daniel Ebert 65926
Hochschule Aalen - Penetration Testing und Computerforensik

Abstract

In diesem Handout wird die Funktionsweise des Ghidra Decompilers und dessen Einsatz erläutert. Der Ghidra Decompiler übersetzt Maschinencode zu lesbarem C-ähnlichen Code [7]. Ein Reverse Engineer kann den Ghidra Decompiler einsetzen, wenn der originale Source Code für ein Binary nicht zur Verfügung steht. Der Higher-Level C-ähnliche Code hilft dem Reverse Engineer dabei, die Funktionsweise des Codes schneller zu verstehen. Der Decompiler automatisiert nicht die Fehlererkennung. Allerdings vereinfacht der Decompiler den Prozess Fehler zu erkennen für den Benutzer. Ghidra's Decompiler kann Fehler machen [7] [17]. Typische Fehler und wie ein Benutzer diese Fehler korrigieren kann werden in diesem Handout ebenfalls erläutert.

1 Einleitung

Dekompilation ist ein Prozess, bei dem Code von einer Lower-Level Repräsentation zu einer Higher-Level Repräsentation übersetzt wird [6] [5]. Es gibt verschiedene Decompiler, die verschiedene Lower-Level Repräsentationen wie Maschinencode oder Java Bytecode in eine Higher-Level Repräsentation wie C oder Java übersetzen.

Ein Decompiler ist der Ghidra Decompiler [13]. Ghidra ist eine Sammlung von Software Analyse Tools um kompilierten Code zu analysieren [13]. Der Ghidra Decompiler ist eine Komponente dieser Sammlung. Unterstützt werden über 20 Befehlssatzarchitekturen, wie zum Beispiel x86 und ARM [11].

Ghidra's Decompiler übersetzt Maschinencode zu lesbarem C-ähnlichen Code [8]. Das ist nützlich, wenn der originale Source Code für ein Binary Executable oder für Teile eines Binaries nicht zur Verfügung stehen. Während der Dekompilation versucht Ghidra's Decompiler auch Informationen wiederherzustellen, die beim Kompilieren verloren gegangen sind. Zu diesen wiederhergestellten Informationen gehören unter anderem Datentypen [8] und Higher-Level Kontrollflussstrukturen wie If-Else Blöcke und Schleifen [7].

Trotzdem entspricht die Ausgabe des Decompilers nicht dem ursprünglichen Quellcode des Entwicklers [6]. Der Grund dafür ist, dass nicht alle verloren gegangen Informationen wiederhergestellt werden können [6]. Zu den nicht wiederherstellbaren Informationen gehören zum Beispiel die Namen von Variablen.

Das Verwenden eines Decompilers mehrere Vorteile: Ein Reverse Engineer kann dadurch die Funktionsweise eines Programms einfacher verstehen und schneller Fehler entdecken [1]. Außerdem muss der Reverse Engineer keine Assemblersprache lernen [19] und durch den interaktiven Editor von Ghidra's Decompiler können Informationen durch z.B. Kommentare oder Variablennamen notiert werden [8].

Der Rest dieses Handouts ist folgendermaßen aufgebaut: Sektion 2 beschreibt, wie der Ghidra Decompiler funktioniert, das heißt, wie die Übersetzung von Maschinencode zu C-ähnlichen Code implementiert ist. Benutzer von Ghidra's Decompiler müssen beachten, dass der Decompiler bei diesem Übersetzungsvorgang Fehler machen kann. Aus diesem Grund wird in Sektion 3 erläutert, wie ein Benutzer mit dem Ghidra Decompiler arbeitet, um Fehler zu erkennen und um die Dekompilation zu verbessern.

2 Ghidra Decompiler Funktionsweise

Diese Sektion beschreibt, wie der Ghidra Decompiler Maschinencode von einem Binary zu einer Higher-Level C-ähnlichen Repräsentation übersetzt. Dieser Dekompiliervorgang besteht aus 4 Phasen.

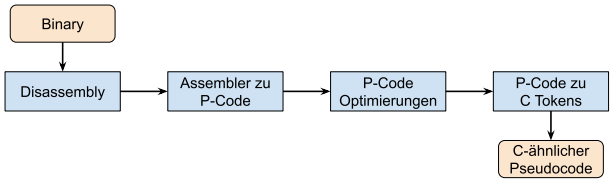

Figure 1: Ghidra Decompiler Funktionsweise Phasen

2.1 Disassembly

In der ersten Phase wird der ausführbare Maschinencode im Binary identifiziert. Eine Herausforderung dabei ist, dass Compiler Daten zusammen mit ausführbarem Maschinencode in derselben ELF Section speichern können. Dies kann zum Beispiel bei Jump Tables der Fall sein [2]. Solche Daten werden auch als Inline Data bezeichnet [2]. Deshalb kann nicht davon ausgegangen werden, dass direkt nach einer Instruktion eine neue Instruktion folgt.

Aufgrund von möglichem Inline Data verwendet Ghidra verschiedene Recursive Descent Algorithmen, um ausführbaren Maschinencode zu identifizieren [21]. Recursive Descent Algorithmen starten an einer gegebenen Maschinencode-Adresse und folgen dann rekursiv dem Control-Flow des Programms [21]. Mit anderen Worten versucht Recursive Descent von einem Einstiegspunkt aus allen möglichen Codepfaden zu folgen um so ausführbaren Maschinencode zu finden. Ghidra verwendet unter anderem den Einstiegspunkt des Programms, die 'main' Funktion, und Cross References um Maschinencode-Adressen für Einstiegspunkte zu finden [21]. Cross References, oder kurz Xrefs, sind Instruktionen und Daten, die über Speicheradressen auf andere Instruktionen oder Daten verweisen [20].

Während dem Recursive Descent wird Maschinencode zur Assemblersprache disassembled/übersetzt. Ghidra verwendet dafür SLEIGH [18]. SLEIGH ist eine Sprache, welche unter anderem die Instruktionen, Register, und Features eines Prozessors spezifiziert [18]. Jeder von Ghidra unterstützte Prozessor ist durch SLEIGH spezifiziert.

Bei Recursive Descent Algorithmen kann es vorkommen, dass nicht der komplette ausführbare Maschinencode gefunden wird. Außerdem können Daten fehlerhaft als 'ausführbarer Maschinencode' identifiziert werden. Der Grund dafür sind Unconditional Branching Instruktionen. Zu dieser Art von Instruktionen gehört zum Beispiel 'jmp RAX'. Für diese Instruktionen kann Ghidra durch statische Analyse nicht immer alle möglichen Ziele des Sprungs ermitteln. Um mehr ausführbaren Maschinencode zu finden, verwendet Ghidra weitere Algorithmen und Heuristiken. Zum Beispiel versucht Ghidra mit Hilfe von Heuristiken typische Funktionsprolog und -epilog zu erkennen, um den Anfang von Funktionen zu identifizieren [21]. Die dadurch gefundenen Funktionsanfänge werden als Einstiegspunkte für weiteres Recursive Descent verwendet [21].

2.2 Assembler zu P-Code Übersetzung

Im nächsten Schritt übersetzt Ghidra Assembler Instruktionen zu P-Code [15]. P-Code ist eine Register Transfer Language (RTL) [15]. RTL kann als architekturneutrale Assemblersprache gesehen werden.

Ghidra übersetzt jeweils eine Assembler Instruktion in eine oder mehrere P-Code Operationen [15]. Diese Übersetzung ist ebenfalls durch die SLEIGH Sprache spezifiziert [18]. P-Code Operationen sind ähnlich wie Assembler Instruktionen. Zum Beispiel gibt es die P-Code Operationen 'INT_ADD', 'LOAD', und 'BRANCH' [15].

P-Code Operationen nehmen eine oder mehrere Varnodes als Eingabe und geben optional eine Varnode aus [15]. Eine Varnode ist eine Verallgemeinerung eines Registers oder eines Speicherbereichs [15]. Maschinencode Instruktionen arbeiten mit Registern und Speicher wie dem Stack und Heap. Im Gegensatz dazu arbeiten P-Code Operationen mit Varnodes. Varnodes können somit als architekturneutrale virtuelle Register bzw. Speicher gesehen werden. P-Code Operationen haben keine Nebeneffekte wie zum Beispiel das Setzen von Status Flags.

Um die Zeit zum Starten von Ghidra gering zu halten, werden nicht alle Assembler Instruktionen sofort beim Start von Ghidra zu P-Code und danach zu C übersetzt. Stattdessen arbeitet der Ghidra Decompiler Just-In-Time. Basierend auf der Auswahl des Benutzers wird eine Funktion dekompiliert. Die erste Instruktion der ausgewählten Funktion wird als Einstiegspunkt betrachtet [16]. Vom Einstiegspunkt aus sucht Ghidra nach allen möglichen Pfaden [16]. Die Instruktionen auf diesen Pfaden werden zu P-Code übersetzt [16].

Das Verwenden von P-Code anstelle von Maschinencode hat mehrere Vorteile für Ghidra's Decompiler. Zum einen müssen die nachfolgenden Analysealgorithmen und Heuristiken nur für eine Sprache entwickelt werden. Ghidra unterstützt verschiedene Prozessoren und verschiedenen Befehlssatzarchitekturen wie x86-64, ARM, und AVR [11]. Ohne diesen Übersetzungszwischenschritt müssten die Analysealgorithmen und Heuristiken an jeden Prozessor und jede Befehlssatzarchitekturen angepasst werden. Durch P-Code muss stattdessen für jeden Prozessor und jede Befehlssatzarchitekturen, welche von Ghidra unterstützt werden soll, eine Übersetzung von allen möglichen Maschinencode Instruktionen zu P-Code implementiert werden. Zum anderen wurde P-Code entworfen, um die Konstruktion von Datenflussgraphen zu erleichtern [15]. Diese Datenflussgraphen werden in den folgenden Analysen benötigt [15].

2.3 P-Code Optimierung

In der dritten Phase wird der P-Code analysiert und nach Lesbarkeit optimiert und transformiert. Diese Phase besteht aus 17 Schritten. Die Schritte 4 bis 9 werden mehrmals wiederholt.

Der P-Code wird optimiert, damit der resultierende Code für den Reverse Engineer einfacher zu verstehen ist [6]. Zum Beispiel werden nicht relevante Details aus dem Code entfernt [6] und Higher-Level Kontrollflussstrukturen wie If-Else Blöcke und Schleifen werden anstelle von 'goto' Anweisungen verwendet [7]. Der optimierte P-Code lässt sich auch besser in C Tokens umwandeln.

1 - Basic Blocks und Control Flow Graph (CFG): Aus

den P-Code Operationen werden Basic Blocks generiert [7]. Diese Basic Blocks werden verwendet, um einen Control Flow Graphen zu erstellen [7] [3].

2 - Analyse von Funktionsaufrufen: Ghidra's Decompiler überprüft, ob bereits Informationen für aufgerufene Funktionen vorhanden sind [7]. Zu diesen Informationen gehören zum Beispiel die Anzahl und Datentypen der Funktionsparameter und der Datentyp des Rückgabewerts. Diese Informationen können zum Beispiel von einem Benutzer spezifiziert worden sein. Es gibt auch Datenbanken mit Informationen über Funktionen von bekannten Bibliotheken wie der C Standard Library und OpenSSL [22]. Die Informationen aus diesen Datenbanken können bei Ghidra mit einem Klick importiert werden [22]. Das ist allerdings nur nützlich, wenn das Programm mit diesen Bibliotheken statisch gelinkt ist.

3 - Hinzufügen von P-Code: Bis jetzt steht dem Decompiler nur der P-Code ab einem bestimmten Einstiegspunkt aus zur Verfügung. Dieser Einstiegspunkt ist die vom Benutzer ausgewählte Funktion. Wenn möglich wird in diesem Schritt neuer P-Code hinzugefügt, damit dem Decompiler in der folgenden Analyse neue relevante Informationen zur Verfügung stehen. Zum Beispiel prüft der Decompiler, ob die Funktion Werte aus dem Speicher liest und ob Ghidra diese Werte bekannt sind [7]. Ist das der Fall, dann werden 'COPY' P-Code Operationen hinzugefügt.

4 - SSA Form: Der bisherige P-Code wird in Static Single Assignment (SSA) Form gebracht. Für die SSA Form darf jede Varnode nur genau einmal zugewiesen werden [3]. Anstatt den Wert einer Varnode zu überschreiben, wird bei der SSA Form stattdessen eine neue Version dieser Varnode erstellt [6]. P-Code in SSA Form vereinfacht die nachfolgende Datenflussanalyse, weil Varnodes nicht wiederverwendet werden [6].

5 - Dead Code Elimination: Dabei wird nicht relevanter P-Code entfernt. Zum Beispiel setzen viele Maschinencode Instruktionen als Nebeneffekt Status Flags, welche für die korrekte Funktion des Programms nicht relevant sind [7]. Außerdem werden unerreichbare P-Code Basic Blocks und Verzweigungsanweisungen, bei denen alle Verzweigungen gleich sind oder die Kondition konstant ist, entfernt [10].

6 - Typ-Inferenz und -Propagation: Ghidra's Decompiler verwendet einen Typ-Inferenz Algorithmus, der jede Varnode mit dem Typ versieht, den er für richtig hält [10] [7]. Für jede Varnode sucht der Algorithmus nach P-Code Operationen, an denen diese Varnode als Operand oder als Ausgabe beteiligt ist [10]. Danach wird überprüft, welchen Datentyp die P-Code Operation als Operand bzw. als Ausgabe erwartet [10]. Zum Beispiel erwartet die P-Code Operation 'FLOAT_ADD' eine Fließkommazahl als Operand. Die Ausgabe ist ebenfalls eine Fließkommazahl. Varnodes, welche bei 'FLOAT_ADD' als Operand oder Ausgabe beteiligt sind, werden daher vom Typ-Inferenz Algorithmus als Fließkommazahl vermerkt [7]. Danach werden diese Typen unter anderem über die P-Code Operationen 'COPY', 'LOAD', und 'STORE' weiter propagiert [10].

7 - P-Code Umschreiben: In diesem Schritt wird der P-Code in Abstract Syntax Tree (AST) Form verwendet [7]. Der Ghidra Decompiler definiert Regeln, wie ein Teil eines AST umgeschrieben werden kann [7]. Das Ziel dieser Transformationen ist die Vereinfachung des P-Codes [7]. Diese Transformationen sind nicht dazu gedacht, den P-Code zu optimieren. Vielmehr sollen die Transformationen einem Reverse Engineer helfen den P-Code und den daraus resultierenden C Code besser zu verstehen [7]. Aus diesem Grund werden in diesem Schritt auch einige der Compiler-Optimierungen rückgängig gemacht.

8 - Dead Code Elimination durch CFG: Basierend auf einer Analyze des CFG wird der P-Code von unter anderem nicht erreichbarem Code, leeren Basic Blocks, und nicht verwendeten Branches entfernt [7].

9 - Wiederherstellen der Kontrollflussstruktur: In diesem Schritt wird die Kontrollflussstruktur der Funktion aufgebaut. Diese Kontrollflussstruktur ist eine Hierarchie [7]. Die Hierarchie besteht aus Higher-Level Kontrollflussstruktur-Objekten wie If-Else Blöcken, Schleifen, und Switch Anweisungen [7]. Der Ghidra Decompiler kann typische Muster für diese Kontrollflussstruktur-Objekte erkennen.

Die Schritte 4 bis inklusive Schritt 9 werden mehrmals wiederholt. Nach jeder Iteration wird der Lower-Level P-Code mehr und mehr zu einer Higher-Level Version des ursprünglichen P-Codes. Die Higher-Level Version kann einfacher in C Tokens übersetzt werden.

10 - Finale P-Code Transformation: Dieser Schritt ist ähnlich wie Schritt 7 'P-Code Umschreiben'. Schritt 7 verwendet teilweise Regeln, welche für die finale Ausgabe nicht ideal sind [7]. Stattdessen werden jetzt nur Regeln angewandt, welche die Lesbarkeit der finalen Ausgabe verbessern [7].

11 - Zusammenführen von Varnodes (Teil 1): In diesem Schritt wird die Anzahl der verwendeten Varnodes verringert. Wenn möglich, werden dabei mehrere Varnodes zu einer Varnode zusammengeführt [7]. Der daraus resultierende P-Code ist ab sofort auch nicht mehr in SSA Form [7]. Zum Beispiel werden Varnodes durch COPY

P-Code Operationen ersetzt [7]. Dabei muss der Decompiler sicherstellen, dass Daten in Varnodes nur überschrieben werden dürfen, wenn diese Daten nicht mehr verwendet werden.

12 - Ausdrücke bestimmen: Im Vergleich zur Assembler Sprache kann eine Higher-Level Sprache wie C komplexe Ausdrücke haben [6], wie zum Beispiel 'a = b + 2 * c'. In diesem Schritt werden die finalen Ausdrücke festgelegt [7].

13 - Zusammenführen von Varnodes (Teil 2): Weitere Varnodes werden zusammengeführt, weil trotz Teil 1 noch zu viele Varnodes existieren [3]. Im Vergleich zu Schritt 11 ist dieses Zusammenführen mehr spekulativ [7]. Die aus diesem Schritt resultierenden Varnodes werden später im C Code zu Variablen und Funktionsparametern.

14 - Type Casts: In Schritt 6 hat der Typ-Inferenz Algorithmus jede Varnode mit einem Typ versehen. Nun werden dem Code Type Casts hinzugefügt, damit die finale Ausgabe syntaktisch gültig ist [7].

15 - Funktionsprototyp: Der Funktionsprototyp für die vom Benutzer ausgewählte Funktion wird festgelegt [7]. Ein Funktionsprototyp besteht unter anderem aus der Anzahl der Funktionsparametern, den Datentypen der Funktionsparametern, und dem Datentyp des Rückgabewerts [16]. Dafür muss der Decompiler bestimmen, welche Register und/oder Stackbereiche verwendet werden, um Funktionsparameter zu übergeben und den Rückgabewert zurückzugeben. Die Typen für alle Register und relevanten Stackbereiche wurden schon in Schritt 6 durch den Typ-Inferenz Algorithmus bestimmt und sind in den Varnodes gespeichert. Die Calling Convention gibt an, welche Register und/oder Stackbereiche für die Funktionsparameter und den Rückgabewert in Frage kommen [16]. Ghidra's Decompiler analysiert außerdem, welche der in Frage kommenden Varnodes gelesen werden bevor sie beschrieben werden. Diese werden als Funktionsparameter angesehen.

16 - Namen für Variablen setzen: Dabei richten sich die Namen der Variablen nach der Verwendung jeder Variable [7].

17 - Finale CFG Änderungen: Abschließend kann die Reihenfolge von Komponenten wie zum Beispiel der Switch Cases geändert werden [7].

2.4 P-Code zu C Tokens

Aus dem optimierten P-Code und den zusätzlichen Informationen über den Funktionsprototyp, dem CFG, den Higher-Level Kontrollflussstruktur-Objekten, und den festgelegten Ausdrücken werden die finalen C Tokens generiert [7]. Beispiele für C Tokens werden in Figure 2 gezeigt.

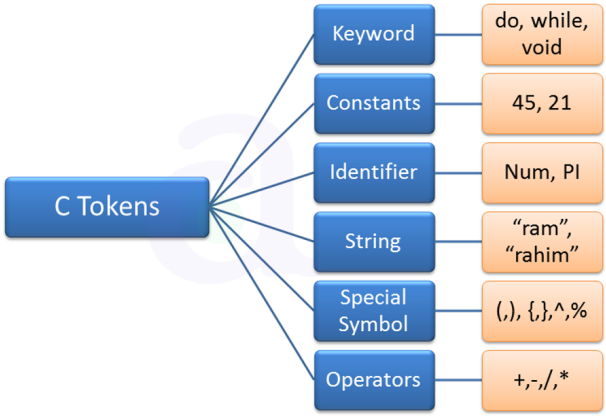

Figure 2: C Tokens Kategorien und Beispiele

3 Typische Probleme bei der Dekompilation

Bei der Kompilierung gehen Informationen verloren [6]. Zu diesen verlorenen Informationen gehören unter anderem Variablennamen, Variablentypen, Kommentare, und die Deklarationen von Datenstrukturen [6]. Es gibt keinen klar definierten Prozess, um die verlorenen Informationen mit 100%iger Sicherheit wiederherzustellen. Das macht Dekompilation nicht unmöglich, aber der Decompiler muss an mehreren Phasen und Schritten im Übersetzungsprozess von Maschinencode zu C Heuristiken einsetzten. Mit anderen Worten muss Ghidra Annahmen treffen. Diese Annahmen können falsch sein. Deshalb kann die Ausgabe des Decompilers falsch sein. Manche Informationen können auch nicht wiederhergestellt werden. Dadurch wird es für den Reverse Engineer schwieriger, den resultierenden C-Code zu verstehen.

Aus diesen Gründen kann der Benutzer die Dekompilation verbessern, indem der Benutzer Ghidra mehr Informationen zur Verfügung stellt. Ghidra weist den Benutzer auf nicht wiederhergestellte Informationen, Fehler, und auf mögliche Fehler hin. Dazu gehört zum Beispiel, wenn Maschinencode nicht disassembled werden konnte oder wenn der Typ einer Variablen oder eines Funktionsparameters nicht wiederhergestellt werden konnte. Diese Hinweise werden in dieser Sektion vorgestellt.

Der Ghidra Decompiler ist interaktiv. Benutzer können sogenannte Program Annotations (dt. Programm Anmerkungen) hinzufügen, ändern, und löschen [17]. Dabei wird die Ausgabe des Ghidra Decompilers dynamisch aktualisiert. Diese Annotations werden ebenfalls in dieser Sektion vorgestellt.

3.1 Instruktionen als Daten identifiziert

Ghidra's Decompiler verwendet für das Disassembly verschiedene Recursive Descent Algorithmen und Heuristiken. Trotzdem können ausführbarere Maschinencodeinstruktionen fälschlicherweise als Daten identifiziert werden. Aus diesem Grund können Benutzer von Ghidra diese Speicherbereiche manuell als 'ausführbarer Maschinencode' markieren. Dadurch können diese Speicherbereiche ebenfalls dekompiliert werden.

3.2 Disassembly schlägt fehl

Wenn der Decompiler Speicherbereiche von Maschinencode zu Assembler Instruktionen übersetzen soll, dann kann dies fehlschlagen. Ghidra's Decompiler gibt in diesem Fall einen 'Bad Instruction' Error aus.

Das Fehlschlagen kann verschiedene Gründe haben. Ein Grund kann sein, dass ein Speicherbereich verschlüsselt oder komprimiert ist und nur zur Laufzeit entschlüsselt oder dekomprimiert wird. Ghidra's Entropie Indikator kann einen Hinweis darauf geben, welche Speicherbereiche verschlüsselt oder komprimiert sind [4]. In diesem Fall könnte man den Code in einer sicheren Umgebung ausführen und den Inhalt des Prozess RAMs speichern und mit Ghidra analysieren. In Figure 3 befindet sich der Entropie Indikator auf der rechten Seite. Je nach der Entropie zeigt der Indikator für einen Speicherbereich schwarz (wenig Entropie), gelb, oder rot (hohe Entropie) an.

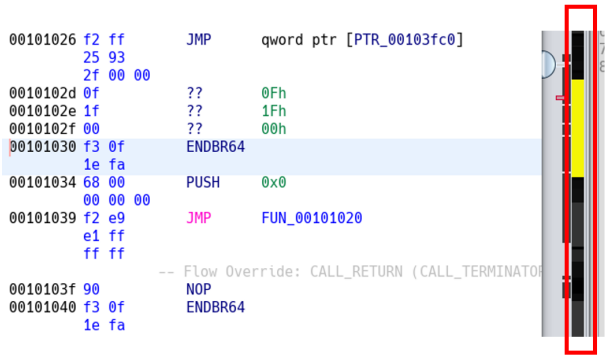

Figure 3: Ghidra Entropie Indikator

3.3 Funktion wird nicht erkannt

Benutzer können eine Instruktion auswählen. Der Decompiler dekompiliert daraufhin die Funktion, zu der die ausgewählte Anweisung gehört [9]. Allerdings kann der Decompiler die Grenzen dieser Funktion nicht immer erkennen [17]. Mit anderen Worten: Die erste und letzte Instruktion dieser Funktion zu bestimmen ist nicht immer möglich.

Ghidra liefert in diesem Fall trotzdem eine Ausgabe [17]. Die erste Instruktion wird ausgewählt, indem der Kontrollfluss von der ausgewählten Instruktion aus zurückverfolgt wird, bis ein Basic Block gefunden wird, dessen eingehende Kanten nur 'CALL' Instruktionen sind [9]. Um die letzte Instruktion auszuwählen, wird dem Kontrollfluss gefolgt bis eine Instruktion mit Terminator Semantik, wie zum Beispiel 'RETURN', gefunden wird [9]. Um den Benutzer anzuzeigen, dass die Grenzen der Funktion nicht ermittelt werden konnten, wird der Hintergrund des Decompiler Ausgabefensters grau und der Funktionsname erhält das Präfix 'UndefinedFunction_' [9]. Benutzer von Ghidra können die Grenzen der zu dekompilierenden Funktion auch selbst festlegen [17].

3.4 Funktion ohne Return

Manche Funktionen sind non-returning. Das bedeutet, dass nach dem Aufruf einer non-returning Funktion der Rest der Aufruferfunktion nicht mehr ausgeführt wird. Non-returning Funktionen sind zum Beispiel Funktionen, die das Programm beenden, wie 'exit()', und Funktionen mit einer Endlosschleife.

Ein Compiler kann non-returning Funktion erkennen und als Optimierung den Rest der Aufruferfunktion entfernen. Dabei wird auch die Instruktion mit Terminator Semantik, wie zum Beispiel 'RETURN', in der Aufruferfunktion entfernt.

Wenn Ghidra nicht weiß, dass eine Funktion non-returning ist, dann versucht der Ghidra Decompiler den Maschinencode nach der 'CALL' Instruktion zu dekompilieren [14]. Dieser Maschinencode ist nicht Teil der Funktion und kann Daten enthalten, wodurch die Dekompilierung fehlschlägt oder Daten fälschlicherweise als Instruktionen interpretiert werden.

Ghidra's Decompiler versucht non-returning Funktion zu erkennen [14]. Zum einen hat Ghidra eine Liste von Funktionsnamen, welche üblicherweise non-returning Funktionen sind [14]. Zu diesen Funktionsnamen gehören 'exit' und 'abort'. Zum anderen werden verschiedene Heuristiken verwendet, die nach Indikatoren suchen, dass eine Funktion non-returning ist [14]. Ein Indikator ist zum Beispiel, dass nach einer CALL Instruktion der Funktionsepilog einer anderen Funktion ist.

Benutzer können einen 'Function Non-return Threshold' angeben. Wenn die Anzahl der Indikatoren für eine Funktion höher als der Threshold ist, dann wird die Funktion als non-returning klassifiziert [12].

Trotzdem kann eine Funktion falsch klassifiziert werden [14]. Aus diesem Grund können Benutzer eine Funktion auch manuell als 'No Return' markieren. Das wird in Figure 4 gezeigt. Wenn eine Funktion fälschlicherweise als non-returning klassifiziert worden ist, z.B. wenn ein zu niedriger Threshhold gewählt worden ist, dann können Benutzer die 'No Return' Markierung entfernen. Wenn eine Funktion nicht in allen Fällen non-returning ist, dann können Benutzer einzelne 'CALL' Instruktionen durch sogenannte Flow Overrides als 'CALL_RETURN' kennzeichnen [17]. Ein Beispiel dazu ist in Figure 5 zu sehen. Der Ghidra Decompiler berück-

sichtigt alle 'No Return' Markierungen und Flow Overrides.

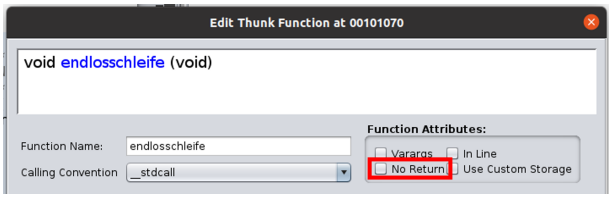

Figure 4: No Return Funktion Markierung

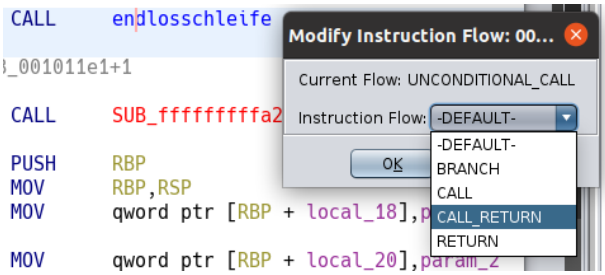

Figure 5: Flow Overrides

3.5 Datentyp

Der Typ-Inferenz Algorithmus kann nicht in allen Fällen die Datentypen aller Variablen, Funktionsparametern, und der Rückgabewerte bestimmen. Wenn ein Datentyp nicht bestimmt werden kann, dann wird der Datentyp in der Decompilerausgabe als 'undefined' angegeben [17]. Benutzer können die Datentypen spezifizieren und Datentypen ändern [17].

Der Typ-Inferenz Algorithmus hat oft Probleme, bei nicht primitiven Datentypen den korrekten Datentyp wiederherzustellen. Zu den nicht primitiven Datentypen gehören unter anderem Structs, Enums, und Klassen. Zum Beispiel werden Structs oft als Array klassifiziert. Die Ausgabe des Decompilers ist dann nicht falsch. Allerdings ist die Ausgabe dann oft nicht gut lesbar. Ein Benutzer kann deshalb auch benutzerdefinierte Datentypen wie Structs, Enums, und Klassen definieren und als Typen verwenden [14].

3.6 Funktionsprototyp

Der Decompiler kann auch Fehler bei der Anzahl der Funktionsparameter machen. Zusätzlich zu den Typen der Funktionsparameter und des Rückgabewerts können Benutzer ebenfalls die Anzahl der Funktionsparameter ändern [17].

3.7 Anti-Decompile Technik

Die zu dekompilierende Funktion kann Techniken verwenden, welche das Dekompilieren erschweren oder verhindern sollen. Zum Beispiel kann der Code eine bedingte Verzweigungsanweisung haben, dessen Bedingung immer Wahr ist [14]. Es kann aber schwierig sein, das statisch herauszufinden [14]. Das bedeutet, dass der Falsch-Pfad durch Dead Code Elimination nicht entfernt wird und ebenfalls dekompiliert wird. Der Falsch-Pfad kann dann Instruktionen enthalten, die den Decompiler absichtlich zu Fehlern führen soll [14]. Zum Beispiel könnte man dort eine 'JUMP' Instruktion einbauen, die zu einer Adresse springt, an der sich Daten befinden. Für solche Fälle können Benutzer das sogenannte Patch Instruction einsetzen [14], um zum Beispiel eine bedingte Verzweigungsanweisung durch eine 'JUMP' Instruktion zum Wahr-Pfad zu ersetzen.

Allerdings kann es auch für Benutzer schwierig sein, toten Code zu erkennen. Dafür könnte man den Code in einer sicheren Umgebung ausführen und analysieren, welche Basic Blocks ausgeführt werden.

3.8 Data Mutability

Data Mutability beschreibt, wie sich Daten in einem Speicherbereich während der Ausführung verändern können [17]. Ein Speicherbereich ist eine einzelne Variable oder ein größerer Speicherblock [17].

In Ghidra werden die Daten in einem Speicherbereich entweder als 'Normal', 'Constant', oder als 'Volatile' klassifiziert [17]. 'Normal' bedeutet, dass sich die Daten ändern können [17]. Das passiert in diesem Fall allerdings nur, wenn eine Instruktion explizit die Daten überschreibt [17]. 'Constant' Daten ändern sich nicht [17]. Als 'Volatile' klassifizierte Daten können sich ändern, auch wenn keine Instruktion die Daten explizit überschreibt [17]. Das ist zum Beispiel bei Shared Memory oder bei Memory Mapped I/O der Fall.

Wenn der Ghidra Decompiler mit Daten arbeitet, die nicht als 'Volatile' klassifiziert sind, kann er bestimmte Vereinfachungen nicht einsetzen [14]. Benutzer können die Mutability von Speicherbereichen ändern [17].

3.9 Probleme im Code

Ghidra's Decompiler kann Variablen mit dem Präfix 'in_' und 'unaff_' versehen [14]. In diesem Fall denkt der Decompiler, dass die Variable einen nicht initialisierten Wert enthält, der verwendet wird [14]. Mit anderen Worten: Die Variable wird verwendet, bevor in sie geschrieben wird.

Das kann ein Fehler im Programm oder ein Fehler des Decompilers sein. Eine mögliche Ursache ist, dass ein Funktionsprototyp falsch ist [14]. Zum Beispiel könnte ein Rückgabewert eine falsche Größe haben [14].

Die Ausgabe des Decompilers kann P-Code enthalten. In diesem Fall konnte der Decompiler diesen P-Code nicht in C Tokens umwandeln [14].

References

[1] Dennis Andriesse. *Practical Binary Analysis*. USA, 2018.

[2] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600, Austin, TX, August 2016. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse.

[3] Chase Kanipe. Ghidra decompiler design writeup, 2021. URL: http://chasekanipe.com/writeups/ghidra.pdf.

[4] Danny Quist. Reverse engineering gootkit with ghidra part i - enable entropy visualization, 2019. URL: https://dannyquist.github.io/gootkit-reversing-ghidra/.

[5] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, USA, 2011.

[6] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. USA, 2005.

[7] Ghidra Developers. Decompiler analysis engine, 2021. URL: https://github.com/NationalSecurityAgency/ghidra/blob/da94eb86bd2b89c8b0ab9bd89e9f0dc5a3157055/Ghidra/Features/Decompiler/src/decompile/cpp/docmain.hh.

[8] Ghidra Developers. Decompiler intro, 2021. URL: https://fossies.org/linux/ghidra/Ghidra/Features/Decompiler/src/main/help/help/topics/DecompilePlugin/DecompilerIntro.html.

[9] Ghidra Developers. Decompiler window, 2021. URL: https://fossies.org/linux/ghidra/Ghidra/Features/Decompiler/src/main/help/help/topics/DecompilePlugin/DecompilerWindow.html.

[10] Ghidra Developers. Ghidra github coreaction.hh, 2021. URL: https://github.com/NationalSecurityAgency/ghidra/blob/4e16b3aa3a649b87a54a6e43a5c01360fd255a83/Ghidra/Features/Decompiler/src/decompile/cpp/coreaction.hh.

[11] Ghidra Developers. Ghidra processors, 2021. URL: https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Processors.

[12] Ghidra Developers. Github checknonreturningindicators funktion, 2021. URL: https://github.com/NationalSecurityAgency/ghidra/blob/da94eb86bd2b89c8b0ab9bd89e9f0dc5a3157055/Ghidra/Features/Base/src/main/java/ghidra/app/plugin/core/analysis/FindNoReturnFunctionsAnalyzer.java#L486.

[13] Ghidra Developers. Github ghidra readme.md, 2021. URL: https://github.com/NationalSecurityAgency/ghidra.

[14] Ghidra Developers. Improving disassembly and decompilation, 2021. URL: https://ghidra.re/courses/GhidraClass/Advanced/improvingDisassemblyAndDecompilation.pdf.

[15] Ghidra Developers. P-code reference manual, 2021. URL: https://ghidra.re/courses/languages/html/pcoderef.html.

[16] Ghidra Developers. Program annotations affecting the decompiler, 2021. URL: https://fossies.org/linux/ghidra/Ghidra/Features/Decompiler/src/main/help/help/topics/DecompilePlugin/DecompilerAnnotations.html.

[17] Ghidra Developers. Program annotations affecting the decompiler, 2021. URL: https://github.com/NationalSecurityAgency/ghidra/blob/3b867b3444afa0d979aab467dcfa901ee6585e14/Ghidra/Features/Decompiler/src/main/help/help/topics/DecompilePlugin/DecompilerAnnotations.html.

[18] Ghidra Developers. Sleight, 2021. URL: http://ghidra.re/courses/languages/html/sleigh.html.

[19] Ilfak Guilfanov et al. Advantages of the decompiler, 2021. URL: https://hex-rays.com/decompiler/.

[20] Omar Darwish. What're you telling me, ghidra?, 2020. URL: https://byte.how/posts/what-are-you-telling-me-ghidra/#cross-references.

[21] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask, 2020. arXiv:2007.14266.

[22] threatrack et al. Ghidra function id dataset repository, 2021. URL: https://github.com/threatrack/ghidra-fidb-repo.