TUHH
Hamburg University of Technology

Diplomarbeit

Arne Wichmann

Binary Analysis for Code Reconstruction of Control Software

11. Oktober 2012

supervised by:
Prof. Dr. Sibylle Schupp
Prof. Dr.-Ing. Andreas Timm-Giel
Dipl.-Ing. Andreas Dierks

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

TUHH STS
Software
Technology
Systems

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 11. Oktober 2012

Arne Wichmann

Contents

List of Figures

List of Tables

Introduction

One of the tasks of a maintenance, repair, and overhaul (MRO) shop is testing of control systems. The shop uses information from maintenance manuals. These manuals may contain gaps. One possibility to close such gaps is the reconstruction of the code that runs in the control system.

This thesis investigates binary analysis for code reconstruction of control software. The original code of a control software is compiled into a binary that is executed on the hardware. The absence of debug information in a binary introduces disambiguities that need to be resolved to reconstruct the code of a system.

An ideal code reconstruction must uncover all statements that are contained in the union of all possible execution traces. Unreachable code that a non-ideal-optimizing compiler may have produced may be uncovered. Bytes in the executable that are not part of the code must not be decoded into instructions.

The linear sweep approach to disassembly is unable to distinguish between code and data in an executable. It marks the minimal constrains for reconstruction: It is necessary to use techniques that follow the control-flow of the code from an initial entry point.

Recursive traversal disassemblers process and uncover the control-flow graph by resolving the targets of jumps and calls. Indirect jumps and calls cannot be resolved without data-flow information in general. Recursive traversal disassemblers make use of heuristics to resolve such indirect jumps.

Iterative disassemblers use a different approach to calculate the targets of indirect jumps and calls. They alternate rounds of disassembly with data-flow analyses on the so-far uncovered control-flow graph. Thereby they can calculate the actual targets of the indirect jumps and calls.

This thesis develops cf96, a lightweight, recursive-descend disassembler for the MCS-96 architecture, which is able to automatically recover the switch idiom of the architecture's compiler.

It compares cf96 with IDA, the leading, commercial, recursive descend disassembler and Jakstab, an academic, iterative disassembler on two input binaries (a test file and an example system's image) using metrics that describe the precision of the reconstructed code. In this comparison cf96

is on par with IDA Pro concerning the precision of the reconstructed code. Jakstab is more precise than both of the tools on the test binary, but cannot reconstruct the whole executed code of the example system's image.

This observation allows the conclusion that the lightweight approach to code reconstruction is sufficient.

Altogether this thesis makes the following contributions:

- cf96, a lightweight recursive descend disassembler.
- IDA Pro MCS-96 CPU module, with support to add an offset to all data accesses.
- Jakstab extension to MCS-96 architecture.
- Case studies, to assess the three different tools using a specially generated test binary and a system image of an example control system.

This thesis is divided into five chapters.
The motivation to code reconstructions and the problems (both general and special) of different approaches (linear, recursive, iterative) are described in chapter 1.
Chapters 2 and 3 describe the foundations on which code reconstruction can work, the different reconstruction approaches, and how to tag the different functions and data locations in the reconstructed code using available informations about the control system.
Chapter 4 presents cf96, the lightweight recursive traversal approach to code reconstruction and describes concepts how to give evidence that the reconstructed code is not faulty.
Chapter 5 evaluates cf96 by comparing the reconstructed code with the output of IDA Pro and jakstab. It describes a system and compiler, which are used in a code reconstruction experiment, the switch heuristic extracted from the compiler, extensions of the existing tools, and the result of the code reconstruction experiment.

1 Binary Analysis Problems

This chapter gives an overview of the purpose of binary analysis and the typical problems that need to be solved. It introduces the assembly language (MCS-96 assembly) that is used in this thesis.

Different kinds of analyses are able to cope with some or all of the theoretical problems. The most important sensitivity criteria are control-flow and data-flow sensitivity.

1.1 Motivation

The first and very reasonable question is: Why analyze binaries and not the source code? Working on binaries or assembly language is not widely practiced, but necessary in case of unavailable source code, untrusted tool chains, or hidden or unintended functionalities.

Missing Source Code The source code of a system is usually not accessible except for the original manufacturer. Source code can be lost over time or inaccessible for other reasons.

To assess the compliance of systems with their specification or to analyze faults it can be necessary to know how the system internally works.

Legacy software, where only executables exist, force the analysis of the binaries to recover information about the functionality and allow the resumption of development of the software.

In case of missing source code it is important to transform the machine code into a more human readable form. The goal is to either help the reader with annotations of the assembly language or to transform the program into a representation in a higher programming language (for example C).

Depending on the purpose of the analysis, it can be necessary to recover all of the executed code to be sure not to miss any functionality. The introduction of spurious code that does not exist in the real system is acceptable because the analyst can manually remove it.

Malware Detection Another motivation to analyze binaries stems from the security community, which wants to know about the use of exploitable software as part of an attack vector. Exploitable code can be introduced by programming errors, hidden, or unintended functionality, resulting from effects introduced by the compilation process.

For malware detection, it is not necessary to transform the whole program into an understandable representation. It is sufficient if the analyst gets hints, which program locations to look at in particular. False alarms are acceptable.

Program Reasoning As a third motivation, it is more efficient to analyze the executable and not the source code in some cases. In general, the compiler can reduce the possible control flow during optimization, helping to reduce the state space that needs to explored in an analysis. Analyzing the executables and not the source code also removes the necessity to trust the compiler/tool chain.

The introduction of spurious code by the code reconstruction leads to a larger state space for the additional analysis and may introduce false information.

All of the motivations share one common subproblem: Disassembling or the recovery of the possibly executed machine code. While it is a trivial task to convert bytes into a single statement, the reconstruction of code from a binary poses a challenge because the statements are not saved as a single linear sequence in the binary.

In this thesis the software of a control system is analyzed to answer high-level questions about it, concerning the existence of functionality in the code or the check whether certain specified requirements are met, posed by the industry.

To answer these questions it is necessary to recover the source code from the binaries. After the reconstruction of the source code it is analyzed to answer the industry question.

This thesis contributes an evaluation of two recursive traversal and one iterative approach to reconstruct code from binaries.

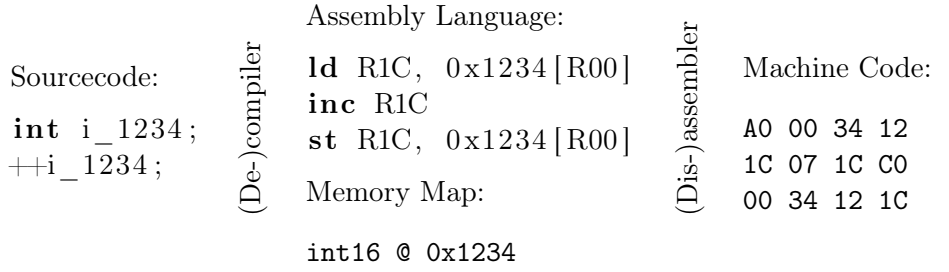

Figure 1.1: Basic Information Flow During Decompilation

1.2 Compilation and Decompilation

To understand binary analysis it is important to know how binaries are generated and what information is of interest for a binary analysis.

Compilation Compilation is the chain of transformations of high-level source code down to low-level machine code.

Compilation happens in a tool chain, where the source code is stepwise processed into an assembly representation by a compiler. The assembly is then processed by an assembler into the machine code. Finally, one or more chunks of machine code and sections of constant data are processed by a linker to form an executable.

During compilation, information can be lost in each step. High-level concepts get transformed to their implementation and the original representation of the high-level concept is no longer available. The challenge of binary analysis is to regain the lost information or to construct approximations to these informations to help further analyses to provide more precise results.

Figure 1.1 lists a simplified example of representations in the tool chain. The high-level C source code declares an integer variable i_1234 and increments its value. After compilation and resolution of symbols the same operation is represented by three instructions and a memory map information about the location of the variable. Finally, the assembler produces machine code, representing the assembly instructions.

Decompilation Decompilation is the process of transforming machine code back to high-level source code, reconstructing lost abstract concepts if possible or constructing abstract concepts similar to the original ones used in the original source code.

To perform its task, the disassembler needs information about the executed statements as well as the memory access, but this information must be explicitly given in the debug information or reconstructed by analyzing the binary.

The disassembler recovers the assembly language representation from the machine code. A general disassembler is unable to recover the same representation of the machine code in the assembly language because of the problems described in section 1.4. The general problem is that the binary does not contain a linear sequence of statements to be executed, but may also contain data, padding, and the control structure usually contains jump that break the linear structure in the code parts.

Another goal of decompilation is the recovery of type information. A type recovery analysis runs into disambiguities caused by similar or identical implementations of different high-level types. It is generally impossible to recover the original types, because this information is lost during compilation and can only be approximated.

Compilation will always reduce the abstraction level in every step. Decompilation must therefore recover approximations to former abstractions that are as similar as possible to the original one.

1.3 Assembly and Machine Code

This section introduces assembly language and its translation to machine (hex) code, using MCS-96 assembly as example.

Assembly language is a programming language designed to represent machine instructions. Because of the difference in machine instruction sets, an assembly language is machine-dependent. Even on a single architecture it can be split up into different dialects, used by different development software. A famous example of such dialects is the splitting of x86 assembly language into Intel syntax and AT&T syntax.

The general concepts in syntax and semantics as well as conversions to machine instructions are shown in the next subsection using MCS-96 assembly language (see [13]).

1.3.1 Syntax

Generally, every assembler instruction is built from an operator followed by a list of operands (may be empty). The operator can be modified by suffixes. The suffix of an operator is a list of properties of an operator.

asm:
    operator operands
operator:
    mnem suffix
operands: one of
    op
    operands , *op*
suffix: one of
    `add`
    `sub`
    ...
suffix: combinations of
    `u`
    `c`
    ...

```
add     op1 , op2 , op3
ld      op1 , op2
push    op1
ret
```

Figure 1.2: Assembly Language Syntax

Examples of operator modifying suffixes are "u", converting an operation to its unsigned form, or "c", for an operation to take the state of the carry flag into account (**mulu** = multiply unsigned, **addc** = add with carry).

Instruction operands have a certain direction, determining which operands are to be considered as source of an operation and which are the destination. There are two major conventions, to write down the destination either first or last. In this work, like in Intel development tools, the destination is written first, fixing the reading of the operands from right to left in terms of data-flow direction.

For binary operations that calculate a result from both operands, the first operand is used both as argument and as destination of the operation.

Figure 1.2 shows examples of the addition operation (**add**), a load from memory (**ld**), pushing a variable on the stack (**push**), and returning from a function (**ret**) to illustrate the grammar.

1.3.2 Operations

The main component of an assembler instruction is the operator. Each operator has a unique opcode, given in an opcode table (see [13]). Instructions can be grouped according to the functionality they provide and the

concepts they implement. Typical groups include instructions that calculate (Arithmetic and Logic), actively change the program counter (Jump), copy data (Memory), or operate on a stack. We briefly explain each group in this section.

Memory Instructions Memory instructions have no calculation effect, they only copy data. Typical memory instructions are "**ld** dst, src" load from memory, "**st** src, dst"[1] store to memory. Usually there are more complex copy instructions that allow the transfer of more than entity at once (here: **bmov**).

Arithmetic and Logic Instructions The main calculation instructions fall in the category of arithmetic or logic instructions. These are, for example, arithmetic (**add**, **sub**, **mul**, **div**), logical (**and**, **or**, **xor**), simple arithmetic (**inc**, **dec**) and bit/arithmetic shift (**shl**, **shr**, **shra**).

Flag	Name	
Z	Zero	result equal to zero
N	Negative	result is negative
V	oVerflow	result is outside the range for the destination
VT	oVerflow Trap	sticky version of V
C	Carry	arithmetic carry or last bit shifted out
ST	STicky bit	sticky version of C during shift

Table 1.1: MCS-96 Condition Flags

These instructions together with the "**cmp**, **setc**, **clrc**, **clrvt**"[2] instruction modify the processor internal state flags stored in the program status word (see table 1.1).

Jump Instructions All of the instructions that carry out simple direct modifications of the program counter belong to the jump category. Control-flow instructions are either conditional or unconditional. Conditional instructions (j...) execute their jump according to the valuation of the processor's state flags. Table 1.2 shows the possible conditions that are available for jumps on MCS-96. Unconditional jumps (**sjmp**, **ljmp**, **br**) always alter the control flow to their target.

[1] **st** is the only instruction to break the operand direction rule.
[2] **cmp** is substraction without storage of the result.

Name	Logic	
C	$C = 1$	carry
NC	$C = 0$	no carry
E	$Z = 1$	equals
NE	$Z = 0$	not equals
V	$V = 1$	overflow
NV	$V = 0$	no overflow
ST	$ST = 1$	sticky
NST	$ST = 0$	no sticky
GE	$N = 0$	greater or equal
LT	$N = 1$	less
GT	$N = 0 \wedge Z = 0$	greater
LE	$N = 1 \vee Z = 1$	less or equal
H	$C = 1 \wedge Z = 0$	higher
NH	$C = 0 \vee Z = 1$	not higher
VT	$VT = 1$	overflow trap ; clear VT
NVT	$VT = 0$	no overflow trap ; clear VT

Table 1.2: MCS-96 Jump Conditions

Stack Instructions Simple stack instructions (**push** and **pop**) just allocate or free space on the stack and write their data to the stack or read it again.

There are special stack instructions, **pushf**/**popf**, that save and restore the program status word and disable/enable the system interrupts, and **pusha**/**popa** instructions that also operate on the program status word and the system interrupt mask.

Callstack Instructions The callstack instructions (**call** and **ret**) operate on the same stack in memory as the simple stack instructions, but perform a more complex task. They save or retrieve the instruction pointer on the stack, and then perform a jump to their target (**call**) or a jump to the retrieved instruction pointer (**ret**).

Special Instructions Special instructions perform operations controlling the processor hardware or do not fit into any of the other categories.

Interrupt handling in the processor is supported by the (**ei**/**di**) enable and disable interrupt instructions. For software interrupts, there also exists a "**trap**" instruction that behaves like a call to a fixed address (`0x2010`) and is used to implement functionality like bios/os calls.

The processor hardware is controlled using instructions like "idlpd" (idle powerdown), "**rst**" (reset), or "**nop**"/"**skip**" (no-operation) instructions.

Instruction Data Width

Arithmetic operations are performed using the processor word width (MCS-96 word: 16 bit). It is possible to modify these instructions with a byte suffix (b) limiting the operation in its width. Some operations also support double word (l) operation.

The access of operands is performed with the width of the operation to be performed. Multiplication and division access their destination with twice the width of their operands, storing either their larger result (multiplication) or a pair of their result and the remainder (division).

For explicit type converstion the instructions **ldbse**/**ldbze** (load byte sign/zero extend) or ext (sign extension) can be used.

Many architectures support access to one and the same register using different data widths. It is also common to allow subaddressing of parts of the register. This possibility leads to register aliasing where updates to a register can kill parts of another register that was not obviously part of the operation. Any data-flow analysis needs to cover this fact to perform a sound analysis.

1.3.3 Addressing Modes

Every operand of an instruction uses a specific addressing mode, addressing a register or memory or using operations to calculate memory locations.

All except the last of the operands for a given instruction use register direct addressing. The last operand in the list can use one of the following addressing modes.

Mode	Example		
direct	**add**	R1C, R1E	R1C += R1E
indirect	**add**	R1C, [R1E]	R1C += ∗(R1E)
indir. + incr.	**add**	R1C, [R1E]+	R1C += ∗(R1E++)
indexed	**add**	R1C, 0x1234[R1E]	R1C += 0x1234[R1E]
immediate	**add**	R1C, 0x1234	R1C += 0x1234

Table 1.3: Assembly Language Addressing Modes

Direct Register direct operands address one register of the CPU, and directly access its content.

Immediate Immediate operand do not access memory or registers, but directly use the operand as a value in the operation.

Indirect Indirect operands use the contents of a register as an address to access the actual content. As a specialization the register might automatically be incremented after the access.

Indexed Indexed operands combine immediate and indirect operands, by adding the immediate offset to the register value and then using this address to access the content. The MCS-96 architecture supports the use of short (byte) and long (word) immediate parts of the indexed operands.

1.3.4 Translation to Machine Code

Every assembly instruction can be translated without loss of information to and from a machine code instruction.

There are two major conventions how to organize a machine code: a variable length list of bytes and a fixed (word size) instruction length.

The fixed instruction length usually enforces a fixed alignment of the interpretation of instructions. Every instruction needs to be encoded into the number of bits available per instruction. Usually the addressing modes for one instruction are not as flexible as they are for variable length instructions.

On machines with variable length instructions, the translation of assembly language to machine code is a step-by-step transformation. Each operator is represented by a number, called the opcode. If there are operands, their addressing mode needs to be specified, before they can be encoded. Specification of the addressing mode can be done within the opcode or by prefixing each operand. The opcode table in [13] defines a unique relation between operators, addressing modes, and opcodes. For instructions without operands a lookup of the opcode from the opcode table is sufficient. If operands exist, they are encoded one by one according to their specification.

Figure 1.3 shows the encoding of an add instruction in 3-operand form. The first step is to look up the opcode and addressing mode in the opcode table. The second step is to encode the special operand using the addressing mode. The last step is to list the remaining register operands.

add R1C, R1E, 0x0123[R20] ⟺ `47 21 23 01 1E 1C`

```
              add R1C, R1E, 0x0123[R20]
               |   |    |      | |  |
47: 40 (add) -/ +7 (indexed) -/-/--/
21: 20 (R20) + 1 (long ind.) -/-/--/
23: 23 (low addr) |----|------|-/
01: 01 (high addr) ----|------/
1E: 1E (R1E) -----|----/
1C: 1C (R1C) -----/
```

Figure 1.3: Assembly Language to Machine Code

Add in its three operand form is encoded with the opcode `0x40`/(mask: `0xF8`). In the three low bits of the opcode, the addressing mode (`0x7` = indexed) is specified. Together, this results in the first byte 0x47.

Next, the special operand is encoded: `0x0123[R20]` is a long indexed operand. The long indexed operand is encoded in the lowest bit (`1`) of the register part. The resulting bytes are therefore `0x21`, `0x23`, `0x01`.

The two register direct operands are encoded as `0x1E` and `0x1C`. The whole instruction is represented as `0x47`, `0x21`, `0x23`, `0x01`, `0x1E`, `0x1C`.

1.3.5 Symbols & Debug Information

Assembly language is usually written by a compiler or a human being and supports symbolic names for entities such as start addresses of functions, variables, and constants. The normal transformation process from assembler to machine code resolves these symbols and generates a mapping of symbols to addresses.

This information helps recovering source code as well as high-level concepts in the source code of an executable, but it is not generally present in the final executable. It is only saved in a memory map external to the executable or as debug information in the executable.

Debug information Debug information gives correlations between machine instructions and their prior position in the source code and information about high-level types of memory positions. Using debug information it is usually possible to find the start of code blocks from this information to separate code and data.

The software security community does not necessarily trust available debug information because it may be forged to mislead an analysis or the high-level concept might hide details of interest.

Stripped Binaries In many cases debug information is no longer available in production code, either to save space or to prevent disassembly.

Altogether a disassembly method cannot assume the availability of a memory map or debug information to recover the executed code of a binary.

In this thesis, a memory map of the system is provided in the system documentation, but the executable does not contain any known debug information and the code must be recovered using specialized techniques.

1.4 Code Recovery

The term "code recovery" denotes the extraction of every possibly executed statement from a binary. A possibly executed statement is an assembly language statement that could be executed by a real system running the binary.

It is theoretically possible to only save instructions in a binary, without any padding or gaps. Real-world binaries, however, usually contain memory images that do not only contain instructions, but also constants, space or initialization for variables and padding.

Because of this, a simple linear decoding of instructions is not sufficient and the discovery of start bytes of instructions is necessary. Such discovery goes beyond a mere syntactic pattern matching, but requires, as we will argue, a semantically sensitive analysis.

In the following the common problems of code recovery are described in more detail. The sections 1.4.1 and 1.4.2 cover the general problems that every code recovery strategy has to cope with; more complicated situations that need to be covered by specialized analyses are discussed in sections 1.5.1 to 1.5.3. Section 1.4.3 discusses individual memory architectures, which complicate the analysis. Those are often found embedded systems.

1.4.1 Program and Data Ambiguity

In binaries, a mixture of machine code interleaved with constants or data is quite common.

An analysis that is not sensitive to the semantics of the instructions it decodes can only traverse its byte stream trying to disassemble every chunk

	Address	Bytes	Assembly Language
A	0x122A	64 1C 1C	**add** R1C, R1C
	0x122D	A7 1D 34 12 1C	**ld** R1C, 0x1234[R1C]
	0x1232	E3 1C	**br** R1C
B'	0x1234	38 12 3C	**jbs** 0, R12, 0x1234+0x3C
	0x1237	12 B1	**notb** RB1
	0x1239	1C	??
	0x123A	17 F0	**incb** RF0
t	0x1234	38 12	**dw** 1238
	0x1236	3C 12	**dw** 123C
B	0x1238	B1 1C 17	**ldb** R1C, 0x17
	0x123B	F0	**ret**
C	0x123C	B1 1C 04	**ldb** R1C, 0x04
	0x123F	F0	**ret**

Figure 1.4: Misalignment in Binaries due to Mixture of Program and Data.

into an instruction, even if it really refers to data. Because instruction sets/opcodes are packed quite densely and only few constraints for operands exist, such behavior usually decodes data into bogus instructions.

Figure 1.4 illustrates a sematically sensitive and insensitive decoding of instructions. Block (A) fetches the address of a second block (B or C) from a table (t) and jumps to this block. Block (B') is the raw decoding of the bytes starting at address 0x1234 and contains nonsense instructions. Block (B) and (C) are the table's jump targets and return values (0x17 and 0x4) from the function.

Block (A) of instructions is successfully decoded in both cases. The second block (B') is decoded wrong by semantically insensitive analyses, because the analysis misses the semantics of the jump instructions at the end of the first block (A). Block (B) and table (t) show a semantically sensitive disassembly of the same bytes, taking care of the jump semantics and skipping the interleaved data. Both analyses correctly identify block (C) as successor of block (B') respectively block (B).

Misalignment

Instruction decoding is called misaligned, if the interpreted bytes are part of an instruction, but the decoded instruction start bytes do not align with the real start bytes.

Due to the high density of instruction sets, valid looking interpretations are possible that have an offset to the real one. Such an interpretation contains nonsense instructions, but each instruction follows the syntax of the machine code. If no syntax errors result, it is only possible to detect such misinterpretations by reasoning about the decoded instructions' semantics. It might even be the case that after a few bogus instructions the instruction flow realigns and further instructions are decoded correctly. [21] describes this phenomena as self-repairing disassembly.

As we have already seen in section 1.4.1 interpretation of data might lead to bogus instructions, but the interpretation of instructions with an offset also produces bogus instructions. Such an interpretation is possible if it continues linear decoding, where control flow performs a jump (for example at the end of blocks or functions), instruction decoding has bugs, the calculation of jump targets produces erroneous results, or the disassembling process has general alignment problems.

Block (B') in fig. 1.4 shows the effect of misaligned interpretation as well as realignment. The bytes at `0x1234` are part of a data table containing addresses. The first three bytes decode to a **jbs** (jump bit set) instruction with two byte operands. The next two bytes decode to **notb** with one operand. The byte `0x1C` at `0x1239` is not an opcode and cannot be decoded. The final two bytes decode to **incb** and one operand.

This case of misaligned interpretation shows a syntax error at address `0x1239` and is therefore easy to spot. It also shows realignement starting at address `0x123C`.

1.4.2 Indirect Code

Indirect jumps cannot be handled without some data-sensitive analysis.

For some instructions control-flow semantic is not only defined by the statements themselves, but also by the valuation of the context during execution.

If a jump instruction contains a target that is not given as an immediate position, but as a reference to a register or memory location, data-flow information needs to be available to calculate possible targets of the jump. If there is not data-flow information available the analysis has to assume a jump to every possible target (every address). It is common to signal these points during the disassembly and not actually assume all possible targets, because this assumption makes the analysis' result unusable.

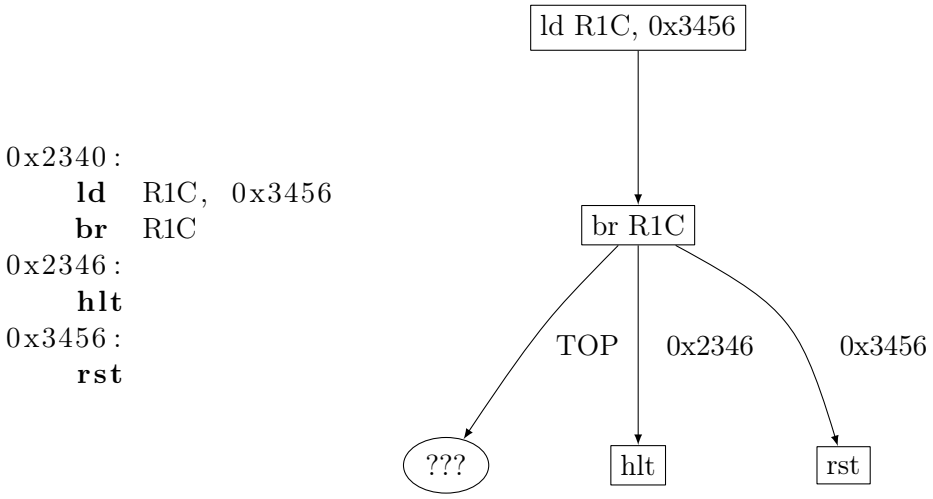

Figure 1.5: Indirect Branch

Figure 1.5 illustrates the indirect jump problem. The first instruction fills the register R1C with the immediate constant `0x3456`. The next instruction is an indirect jump using the value stored in R1C.

A semantically insensitive analysis decodes `0x2346` as the next instruction, just because it is given by the next bytes after the indirect jump. A semantically sensitive analysis without data flow can only assume TOP as target of the indirect jump, assuming every address as the next instruction's start. Only a semantically sensitive analysis with data-flow information can infer, that the only possible target is `0x3456`.

1.4.3 Memory Layout

Any system needs to address its memory. Especially on small systems with limited register sizes but larger memories or on systems with virtual memory, the available memory is split into virtual addresses are translated into hardware addresses.

It is a standard feature of the existing analyses to support the usual or default memory configuration of a system, but the system developer can add modifications to the default layout both in hard- and software and then make it hard to resolve any memory accesses.

If an analysis is unable to take the memory layout into account while resolving the targets of memory access, references to code and data will not be resolvable.

A special problem is the change of memory contents independently from access by the master controller, for example, by memory mapped input/output (MMIO) operations. Using the MMIO technique an independent controller can provide data in a memory area that is reachable by the master controller. If such data access is used by a data-flow sensitive analysis, no assumption on the contents of the memory can be made. It is not sound to assume a fixed value (even if it is the whole range of possible values) for two consecutive read attempts. To perform a sound analysis these reads must return an undefined value.

1.5 Advanced Code Recovery

The following code recovery problems, from the code obfuscation domain, break the usual conventions of an architecture. They are important for analyses, because they can misguide simple code recovery and thereby hide functionality of the code.

In the security and malware sector such obfuscated code is usually hand crafted to specially hide a functionality. For prevention of reverse-engineering, code obfuscation techniques are developed to automatically provide countermeasures (see [19]).

1.5.1 Multialignment

Multialignment or instruction aliasing is the exploitation of instruction misalignment by voluntarily hiding a second stream of instruction in the same memory that is already occupied by the first stream. Malware might use multialignment to hide code from analyses that enforce unique interpretation of data at a given address.

Classical disassembling allows only one stream of instructions with one fixed alignment for a given sequence of bytes. With this assumption, one instruction that occupies some bytes in memory prohibits any additional interpretation of these bytes with an offset. If such an analysis encounters a multialignment, it will probably find a jump instruction, the target of which will jump with an offset into the already fixed stream of instructions. It then cannot interpret the instructions at the jump target, thereby missing some instructions that would be executed on a real machine.

```
0x0000:  B8 00 03 C1 BB    mov eax, 0xBBC10300
0x0005:  B9 00 00 00 05    mov ecx, 0x05000000
0x000A:  03 C1             add eax, ecx
0x000C:  EB F4             jmp $-10

0x000E:  03 C3             add eax, ebx
0x0010:  C3                ret

0x0002:  03 C1             add eax, ecx
0x0004:  BB B9 00 00 00    mov ebx, 0xB9
0x0009:  05 03 C1 EB F4    add eax, 0xF4EBC103
0x0010:  C3                ret
```

Figure 1.6: Multialignment Example from [17]

Figure 1.6 shows an example of a byte sequence containing two streams of instructions. The first stream contains blocks `0x0000-0x000C` and `0x000E-0x0010`. The second stream contains the blocks `0x0000-0x000C` and `0x0002-0x0010`. At address `0x000C` a relative jump instruction alters the control flow to continue at `0x0002`.

A multialignment insensitive analysis can decode the indirect jump at `0x000C`, but cannot resolve its target at `0x0002`, because the bytes are already part of a differently aligned instruction.

An analysis with support for multiple alignments can decode one and the same bytes as part of more than one instruction, allowing the discovery of the second (formerly hidden) sequence of instructions.

1.5.2 Stack (Instruction) Abuse

It is possible to abuse instructions that implement parts of high-level concepts to perform operations the instructions were not intended for.

Using instructions in a way that breaks high-level abstractions that are assumed to hold for the instruction is called instruction abuse.

An important class of abuses concerns the return statement. Using techniques similar to those described in [20], it is possible to abuse the return instruction as a general purpose indirect jump, by manipulating the return address saved on the stack.

Usually every function has a balanced number of pushes and pops leaving no variables on the stack and the topmost element is the return address.

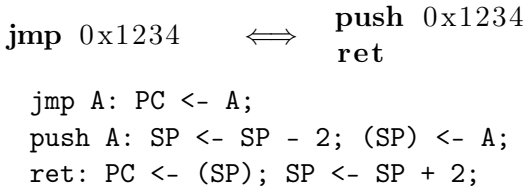

Figure 1.7: Stack Abuse

By pushing a value on the stack, and breaking the convention, the function will return to this value and not to the real return address.

It is important to distinguish this behavior from a normal (optimized) return from a function that can be implemented using manual manipulations of the stack pointer and an explicit indirect branch to the return address. This is a standard compiler behavior and can be detected using heuristics (see section 5.1.4).

If an analysis blindly follows the semantics of the return instruction (leave the current function, and return to the site of the function call), and does not account for changes of what is interpreted as the return address of a function on the stack, it is possible to hide this indirect jump from the analysis.

Figure 1.7 shows a simple instruction sequence consisting of a push and a return instruction, as well as the concrete semantics of the push, return and jump instructions. By convention for normal function use, the return address is stored at the top of the stack. This allows a simple jump-like behavior by pushing the address onto the stack and a return. The return pops the address from the stack and writes it into the instruction pointer. An analysis without semantics for individual stack operations assumes the end of the function at the return statement. Only an analysis covering the semantics of a stack instruction discovers the implicitly encoded jump instruction.

1.5.3 Selfmodification

Any change of a program at its own program code is called selfmodification. While changes to data are common and in general do not pose problems to an analysis, selfmodification poses a big problem for an analysis.

Many architectures allow execution of program code from changeable memory, thereby generally allowing the execution of selfmodified program code. If it is necessary to analyze such code, all of the changes to the

```
ld      R1C, 0x1230[R00]
ld      R1E, 0x1232[R00]
ld      R20, 0x1234[R00]
ld      R22, 0x1236[R00]
add     R1C, R20
addc    R1E, R22
st      R1C, 0x1230[R00]
st      R1E, 0x1232[R00]
```

```
long int x,y;
x+=y;
```

Figure 1.8: Addition of Dwords

executed memory locations need to be tracked in detail, so that the decoding of the memory as instructions is possible.

The loading of libraries at the program startup is usually not considered selfmodification, because the linker information can usually be followed quite well by an analysis.

1.6 Data Type Recovery

A major problem for decompilation is data type recovery. Although it is possible to just express all the low-level data accesses in a high-level language, the reader of the high-level language would be disappointed not to see the use of the abstract concepts that the accesses represent on this level. It is therefore often desired to recover high-level representations of the data access.

The usual types (for C as higher language) that need to be discovered are the width of integer variables, the use of memory to implement software floating point types, the members of structs, and the size of arrays. Targeting an object-oriented language also requires the detection of class hierarchies and class member structures.

Figure 1.8 shows the addition of two 32 bit integer (dword) variables in C and assembler notation. Due to the limit of the arithmetic add instruction to 16 bit (word) width, it is necessary to split the operation into two add instructions, the second of which uses the carry flag of the first (addc). The load and store instructions are split into word size instructions as well.

An Analysis needs to link the independent load and store instructions by understanding the semantics of carry-chained addition.

The reconstruction of abstract type concepts is beyond the scope of this work. It requires data-flow sensitive analyses with complex abstract domains working on a given control-flow graph.

This thesis concentrates entirely on recovering the code of an executable. Two code reconstruction approaches in three tools are evaluated to determine, which code reconstruction problems exist in the analyzed system and how complex it is to adopt the tools to the system.

2 Binary Analysis Theory

A major simplification of code reconstruction is possible if it is known that the code is originally generated by a compiler. Two aspects of compiler generated code are very important.

First, to be linkable the code follows a procedure call standard, which defines how to call or return from a function. If such a function abstraction exists, code reconstructions can exploit it to trivially resolve all indirect jumps that are part of a function's return.

Second, a compiler emits certain idioms for high-level constructs in the original source that are easily detectable and allow simplifications for the calculation of targets of indirect jumps that might be involved.

After the code reconstruction is completed, the code does not contain symbol names. Reading code that only contains addresses and no names is nearly impossible.

It is possible to gather names for the input/output (I/O) boundaries of a code from the actual executing hardware. By systematically propagating these names along the code, functionality clusters can be detected.

Only with these informations it is possible to answer real-world questions.

2.1 Generated Binary Code

To call functions compilers use a calling convention or procedure call standard. These standards allow interoperability of code produced by different compilers, or in different compiler runs in one final executable.

It is also common for compiler-generated code to not be optimally optimized, which leads to the existence of patterns (idioms) in the machine code that represent common patterns of the source code. Typical idioms include function prologues or epilogues and switch implementations with jump tables.

Both the procedure call standard and idioms in the code allow a simplification of the analysis of the machine code.

2.1.1 Procedure Call Standard

Every platform (compiler) uses a certain way to call and pass parameters to its function implementations.

The procedure call standard defines how to call functions and how to receive return data from them. It introduces a call and parameter stack and defines how registers are used during a function call.

To be able to link object files, it is necessary to know the interfaces that the object files provide. The optimization a compiler can perform is therefore limited to the definitions of the interface, keeping optimizations inside single functions.

Many compilers or linkers are not able to proceed with optimization after the linking process, leaving the interfaces available in the final executable.

The common degrees of freedom of a procedure call standard are described in the following.

Stack Every procedure call needs to store its return address and supports a call depth of more than function, forcing the storage of return addresses on some sort of stack.

Parameters While it is possible to store the parameters of a function call in the registers of the machine, this limits the total amount of parameters to the amount of registers available for parameter storage. It is therefore common to also use a stack to store function call parameters (this is usually the same, as the one used for return addresses).

Return Data The return data of a function can be stored in a register or on the stack.

Registers It is necessary to define the general usage of registers in a function call, so that it is possible for the caller, callee, or both to save and restore the registers possibly modified by the other. A special case are interrupt service routines where no assumptions on registers hold in general and all normal function use registers that need to be saved.

Local Variables It is possible to keep the local variables of a function in the registers or to store them on a stack.

Register	Access	Contents
R00	Byte, Word	Static 0
R02 - R17	Byte/Word	Special Registers (IO, Status, Config)
R18	Word	Stack Pointer
R1A	Word	Frame Pointer
R1C - R23	Byte, Word, Dword	Local Variables, Fkt. Return Data
R24 - RFF	Byte, Word, Dword	General Purpose

Table 2.1: MCS-96 Registers

```
fkt(short param1,
    long int param2,
    int param3);
```

???? :param1
high word:param2
low word :param2
param3
return address

← Stack Pointer

Figure 2.1: MCS-96 Stackframe Layout

MCS-96 Register and Stack

We illustrate the architecture terms of the procedure call standard using the Intel MCS-96 architecture stack and register as described in [13].

Table 2.1 shows the 256 available registers on MCS-96. Register R00 is filled with a static constant 0 and registers R02 to R17 are used as special function registers to control and receive status of the controller hardware. R18 is the current stack pointer and is automatically modified by push/pop/call/return instructions. R1A is a function local copy of the stack pointer on function entry and is used to address the local variables on the stack.

R1C to R23 are used as temporary storage inside a function and need to be saved by the callee prior to their usage. R1C is also used to store the return value of a function. An extended version of the procedure call standard allows the extension of this region to R2A. Finally, R24 to RFF are available for general purpose use.

On MCS-96, the stack grows from higher to lower addresses, decrementing the stack pointer to allocate new stack space. Only the function parameters, return address and local variables are saved on the stack. To allocate local variables the stack pointer is further decreased and copied into the frame pointer, thereby allowing indexing of the local variables with positive numbers.

Figure 2.1 shows an example stack frame for the call of a function. It takes three parameters (param1-3) with short, long int and int types. The first entries in the stack frame of it are the parameter variables, followed by the return address of the function. Every parameter entry on the stack is split up into word sized chunks (push/pop can only operate on words). param1 occupies a word-sized chunk with a high byte of undefined value, param2 is split into two words, and param3 fits exactly into one word.

2.1.2 Compiler Idioms and Heuristics

Compilers are developed for a certain platform and usually have patterns stored that are used to transform input constructs into the binary code representation. These constructs usually help to implement the procedure call standard (call/enter, leave a function) as well as the control-flow constructs of the language used (in C, for example, for-loops, switch/case statements).

Switch A common pattern is the implementation of a switch construct from the C language as a combination of a jump address table and an indirect jump instruction. After an optional check of the switch variable bounds, an index into the table is calculated and the jump to the case block is executed. Especially for large and dense switch structures this can significantly reduce the number of condition evaluations that is necessary otherwise.

In the absence of a default case is not uncommon to use the switch variable unchecked in the calculation of the case entry, leading to fetches of invalid target addresses from arbitrary memory outside the table.

As many compilers emit switch idioms in their code, it is a common practise to search for their use and use the table to resolve the targets of the indirect jump involved (see [6]).

Function Prologue Another idiom that can be used on many architectures are function prologues. These usually handle the function entry modifications of the stack. On x86 using gcc for example, prologues consist of a well-recognizable sequence of pushl, movl and subl instructions.

Figure 2.2 shows how such function prologues can be used to recognize the start addresses of functions inside a raw binary, without any other necessary knowledge.

The usage of a known function prologue allows the detection of function start addresses by scanning a file for the known instruction sequence, elim-

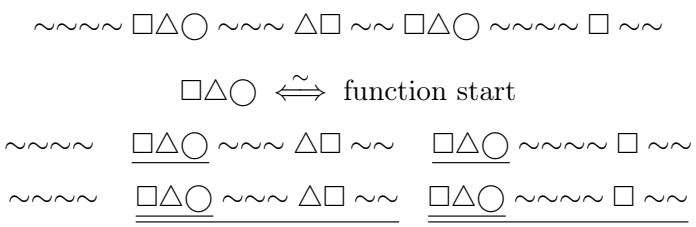

Figure 2.2: Matching of Known Idioms to Instruction Sequences

inating the need for giving entry points into the program explicitly. The MCS-96 compiler does not emit a fixed function prologue.

2.2 Reading and Understanding Code

Reading large amounts of assembly code without symbols or names for functions and data access poses a big problem.

It is generally not possible to understand the meaning of a function just by its code without the knowledge of the data it is working on. Take as example a function that calculates $y = m * x + b$. Without the knowledge of the semantics of y, m, x, and b it is possible to name the function only with a general abstraction (`calc_linear`). If it were known that m and b stem from a calibration read-only memory (ROM) and that x is a value received from an analog to digital (AD) conversion, it would be possible to give the function a more specific name like `ad_apply_calibration`. Conversely, it is hard to find names for data, if it is unknown which operations are performed on it.

In the following several techniques are introduced that support code understanding.

Control Flow Visualization To understand a function, a visualizations of its control flow is of great help. There are two major visualization of control flow: control-flow graphs (CFGs) and call graphs (CGs).

The view of a function as a CFG, shown as a connection of basic blocks, is helpful to understand a function. Simple loops and chains of checks can be spotted this way.

A CG showing the caller/callee relation between functions helps to understand the causal and temporal connection between functions.

Ida Pro ([12]) provides a so-called proximity view that combines part of the call graph with a limited depth with a def/use view grouping the functions in addition by providing information about shared variables.

Compiler Idioms It is possible to recover high-level constructs of the input language by detecting idioms in the source code.

In addition to the idioms described in section 5.1.4, a typical pattern of the for loop emerged in the assembly code.

Information propagation Strategies similar to program slicing can spread available information through the program and name entities according to the known semantics of the slice.

A program slice is a subset of program instructions determined by selecting all of the instructions affecting a certain memory location at a program position ([24]). Program slices are commonly used to follow the propagation of a single value through a program. In this thesis, they are used not to propagate a value but an abstract name of a value along the slice.

By using available information at I/O boundaries of the program, the name of the I/O parameter can be propagated along the functions and variables that are part of the slice. Typically, the I/O information of a program is at least in parts available from the controller manual (interrupt service routines, e.g., serial I/O) or the systems schematic.

Using this information to propagate a name through a program it is possible to name all the variables that depend on the result of an AD conversion with the name of the AD channel. For example, all of the data that depends on the AD value of a pressure sensor can be named with a `pressure_` prefix.

With the described techniques it is possible to uncover enough of the original code's structure and domain semantic to answer real-world questions to the code.

3 Code Recovery Approaches

The ideal result of a code reconstruction is a list of statements containing all the statements that could be executed on the real hardware. More formally, the ideal result is the union of all possible program traces.

An acceptable result, on the other hand, is output that does not miss any possible control flow. In practice the output therefore overapproximates the result. Additional non-executed statements may be contained for several reasons. These instructions are acceptable only if they are originally generated by the compiler.

It is inacceptable if bytes are decoded into instructions that never were intended as such and represent behavior that never could occur on real hardware.

This chapter describes the three major approaches of code reconstruction for binary analysis with their abilities concerning the different classes of results, namely the linear sweep, recursive traversal, and code reconstruction approaches. Linear sweep is unable to fulfill the minimal requirements of code reconstructions. Recursive traversal and code reconstruction can generally cope with compiler generated code and principally calculate acceptable results. Code reconstruction can also handle handwritten obfuscations of cope.

The chapter is summarized by an overview of the available tools.

3.1 Linear Sweep

Linear sweep is the simplest approach to code recovery.

Linear sweep disassembly is completely insensitive to the semantics of the statements it disassembles. The statements are only processed according to their syntax and transformed to their textual representation.

Figure 1.4 from the introduction shows the typical behavior of a sweep disassembler including its problem with constants mixed into the code.

In the presence of usable debug information this approach is successful. It then knows about the start addresses of functions and data in the binary and can process it accordingly.

Linear sweep is not able to address the problems described in section 1.4. It cannot distinguish between program and data, nor can it resolve indirect jumps, detect multialigned instructions, or follow the effects of instruction abuse, and of course it is not able to trace selfmodification.

It still is useful as one of the first tools to process a new binary file, because although the disassembly will be fooled regularly, it might realign and can therefore give a rough idea what to expect in a binary.

One of the most common tools for linear sweep disassembly is GNU objdump ([10]). In this work, the linear sweep disassembler is dis96 ([15]). It reads a file with start and stop addresses of code and data and then traverses the file byte by byte.

3.2 Recursive Traversal Code Recovery

Using platform conventions, a code recovery is possible. The recursive traversal approach only needs information about the semantics of statements directly manipulating control flow.

In the absence of malicious, perfectly globally optimized code it is therefore possible to recover the control flow using additional assumptions of the target platform.

The recursive traversal code recovery models the call graph and recursively follows each function call, recovering function entry points. The intra-procedural control flow is recovered by traversing the jumps inside of a function. The only input needed are the external entry points to the program. The most popular implementor of this approach is IDA Pro (see [12]).

3.2.1 Recursive Traversal

Recursive traversal starts at a program entry point. It decodes statements like the linear sweep approach but limits its interpretation at unconditional jumps (**sjmp**, **ret**, **br**). It marks the targets of call or jump instructions as the start of new functions or code blocks. It then restarts interpretation at these new code points until no new code or functions are detected.

Figure 3.1 shows the CFG of three functions calling each other. An entry point (EP) is defined by one jump instruction external to the three functions. The function starting at (EP) contains a call to (A) and a jump to (EP'). Function (A) calls (B) and then either branches to (A') or returns directly. Function (B) immediately returns.

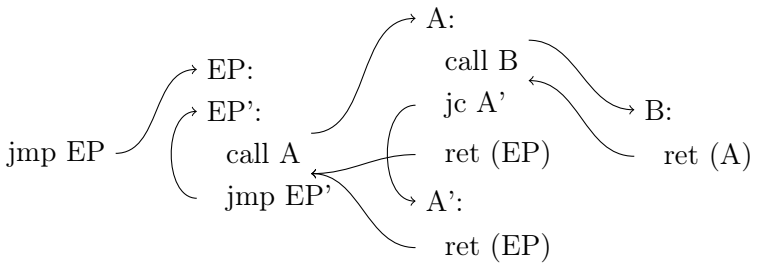

Figure 3.1: Control Flow Graph

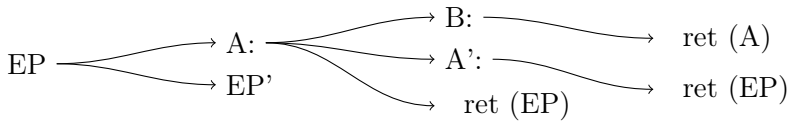

Figure 3.2: Traversal Tree of Code Locations

Figure 3.2 shows the recursive traversal of the control-flow graph from fig. 3.1. The traversal starts at the entry point (EP). The first two locations discovered are (A) and (EP'). Traversal of (A) resolves the locations (B) and (A') and a return to the (EP) function. Traversal of (B) uncovers a return to (A). Traversal of (A') uncovers a return to (EP). The traversal of (EP') does not uncover new code, as the memory is already covered by the initial traversal of (EP). This leaves no call or jump target unexplored and the analysis can stop.

3.2.2 Indirect Control Flow

The resolution of general indirect jumps or calls is impossible for the recursive traversal approach. It does not know about the semantics of any other than the explicit control-flow statements and therefore cannot determine the value of an indirect branch.

In the most general case data-flow analysis is required to know the semantics of all statements and calculate the possible values that a jump instruction may use. We discuss this approach in section 3.3. There are, however, heuristics for two single cases of indirect branches, which allow the recursive approach to successfully disassemble the code.

If indirect branches can be safely assumed to be part of leaving a functions, the determination of the target can be ignored, if normal stack behavior is assumed. If the indirect branch is part of a compiler idiom (especially of a switch/case construct) it is usually possible to simply fetch the possible target values from the table using the known pattern of the construct.

The biggest problem is to determine the length of the table if no explicit masking/limiting of the switched-on value exists, but this problem can be expressed fairly well using code locality assumptions expressed as heuristics (see section 5.1.4).

3.3 Code Reconstruction

It is impossible to completely recover the code and control flow of a binary without the ability to recover the targets of indirect jumps. Data-flow analysis provides a method to calculate such information, but a chicken and egg like problem exists.

To calculate indirect jump targets, data-flow analysis needs a control-flow graph to be able to work, but in the presence of indirect jumps, a control-flow graph can only be calculated if data-flow information is present.

It is possible to combine both techniques and recover control flow, while calculating data-flow information. It is then possible to calculate indirect flow targets, until a fixed point for both control and data flow is reached. Jakstab (see [17]) implements this approach.

To implement this approach jakstab uses a formalism similar to cpachecker (see [18, 4]) with the extension of adjustable precisions for the abstract domains used.

Cpachecker provides a framework that is configurable between a classical data-flow analysis and model-checking approaches by providing operators that implement the merging or separation of two abstract states and a termination check operator.

The actual abstract domains used in the calculation of the analysis can also be specified and combined. Each analysis needs to provide its states, semi-lattice, and concretization function as well as a transfer relation that maps its states using the semantics of the transition specified by a statement.

3.3.1 Statement Semantics

Jakstab defines its analyses on an intermediate language called statement specification language (SSL). It is therefore possible to use the same analysis for different binaries.

This idea and language is already used in different tools (boomerang ([5, 9]), jakstab ([14, 17]) and similar BAP ([2])) and is originally defined in [7]. SSL is modeled closely to a register transfer language, consisting of a memory/register model (especially to model aliasing) and assignments together with the ability to perform elementary arithmetic and logic calculations.

During code recovery every disassembled statement is matched to its SSL representation and the complete analysis is performed on these representations.

As an example of such mappings Figure 1.7 shows the semantic information for some statements given in the microcontroller manual. Each statement's operation is broken up into elementary calculation and an assignment operation, where only elementary operations are allowed and each operand is an abstract memory location.

3.3.2 Bounded Address Tracking

Jakstab provides a specialized abstract domain for code recovery called bounded address tracking.

Bounded address tracking defines a type that is used to describe the location of a tracked entity as well as its contents. Each location can contain a limited set of values, bounded by a configurable constant. If the bound is exceeded stepwise widening occurs.

The type used is a tuple of an abstract memory region (stack, global, allocation, etc) and an offset inside of the region.

The widening on that occurs if the number of states for a given location exceeds the configured bound happens in two step. The first widening step keeps the memory region of the values in tact and summarized the offsets inside of the regions to a top element of the same bit-width as the original values were. The second widening step proceeds to summarize the regions into a top element if the number of individual regions in a state exceeds the bound.

3.4 Available Tools

There are several disassemblers and decompilers on the market. The first selection criteria from a user's point of view is the architecture support or extensibility of the tools.

The second criteria is the aliveness of the tools, for example the time since the last release and availability of support. We briefly evaluate the major tools.

Commercial Tools

IDA Pro The most important commercial tool for source code recovery on the market is IDA Pro ([12]). It grew from an x86-only interactive disassembler to the interactive disassembler with the largest number of supported architectures.

Its interactiveness allows users to correct the analysis if it is wrong or to manually approximate in situation where automatic analysis halts.

For x86, it provides very fine-tuned heuristics to support and automate code recovery far beyond traditional recursive descend recovery. Unfortunately, these heuristics are not usable for other architectures and compilers.

Yet, IDA Pro provides an application programming interface (API) allowing the extension with plugins as well as implementing new CPU architectures.

Codesurfer/x86 Codesurfer x86 is a commercial static analysis tool working on x86 assembly. It uses value-set analysis (VSA) ([1]) as abstract domain and calculates a so-called system dependency graph. Its distinguishing feature is the ability to browse and work with this graph. Codesurfer/x86 uses IDA Pro for disassembly.

Research Tools

Binary Analysis Platform (BAP) Binary Analysis Platform (BAP)[2] is a toolkit to support various analyses on an intermediate language. It includes a tool (toil) that processes assembly language and lifts it to the intermediate language of the toolkit. BAP is currently under active development.

Boomerang Boomerang is a decompiler supporting the reconstruction of C code from x86 and SPARC executables ([9]). Boomerang uses the same

intermediate language as jakstab ([7]). The development stalled in 2006 due to a conflict of interest when the main developers joined a company.

Jakstab Jakstab implements a configurable program analysis variant to recover code. It provides a disassembler and translations to its intermediate language for x86 ([17, 7]). It is designed for extensibility by new architectures, but according to the original author certain hidden assumptions about the x86 language might need to be generalized ([16]). Jakstab is currently under active development.

In this thesis IDA Pro is used as the state of the art commercial benchmark. We also extended Jakstab to the MCS-96 architecture to evaluate the possible gain in accuracy of the reconstructed code.

4 Lightweight Approach

In this thesis an experiment on code recovery is performed with an avionics device. The development environment of such a device suggests that a simplification in the code recovery is possible.

For one, airborne (critical) software is intensely reviewed and follows conventions. The standard about avioncs software development ([22]) enforces extensive documentation and analysis of the resulting software.

Second, the usage of certified compilers and extensive code review allows the assumption of the absence of complex code recovery problems.

It is possible to check the resulting source code for inconsistencies to allow the detection of breaches of the assumptions used.

In the following, a wrapper to a simple disassembler is described that is able to recover the code of a binary following normal stack conventions and produced by a compiler with known idioms. The wrapper uses a generic, architecture independent algorithm that implements a recursive traversal disassembly using the simple linear disassembler. The algorithm relies on heuristics to resolve indirect jumps. Heuristics for the compiler used in the example control system's image are provided in section 5.1.4.

4.1 Wrapping a Sweep Disassembler: cf96

The dis96 disassembler needs manual annotations to disassemble a binary. Writing these annotations for each block of each function is a tedious task.

We therefore developed cf96 as a wrapper around the dis96 disassembler to automate the process of discovering the start addresses of code in an executable, to locate and analyze switch idioms, and to provide alignment directives for ends of code blocks. cf96 uses the recursive traversal code recovery (see section 3.2).

cf96 is able to recover the code of an executable without manual interaction except of the setting of entry points for the analysis.

Command	Description
fName	input file = 'Name'
tXX	string terminator byte (default = 00)
eXXXX	end of dissambly
pXXXX	procedure start
cXXXX	code starts at XXXX
lXXXX	attach a label to address
bXXXX	byte dump
sXXXX	string dump
wXXXX	word dump
aXXXX	alphanum dump
kXXXX	one-line (k)comment for address XXXX
nXXXX	multi-line block comment

Table 4.1: dis96 Disassembly Input File Commands

4.1.1 Foundation: dis96

dis96 is a very simple disassembler for the MCS-96 microcontroller family originally developed by Hengeveld [11] and significantly extended by Johnson [15].

dis96 reads a command file with annotations. It processes a binary file, decoding the file according to the annotations in one traversal. It does not support any automated recovery of the code section in the file. Every code or data section must be annotated in the command file manually.

dis96 keeps a list of addresses that have labels assigned. During instruction decoding each constant is looked up and resolved to its label, if it exists.

Table 4.1 shows the possible input commands to dis96. The list is split into four sections. The first section lists the commands that are specified once per file. The second section specifies start addresses of code blocks in the file that are either functions/procedures (p), code blocks part of functions or raw code blocks (c), or labels inside of code (l). The third section offers different possibilities to annotate stored constants and data access to variables. The last section contains commands to annotate the disassembler output with user-specified content.

Additions to dis96

The memory model of the unit under test makes it necessary to distinguish between code and data access at the same address. We therefore split the label and reference stores of code and data labels. As a consequence, code addresses, introduced as immediate constants in the assembly are no longer automatically resolved to a label.

Because of the new possibility of code and data at the same address it is necessary to process the input file twice, once to disassemble code sections and a second time to annotate constants and variable locations. Since dis96 does not provide an explicit command to annotate the end of a code section, and only supports the switching of code to data on instruction end addresses. We introduce an alignment directive (z) that marks possible transitions of code to data sections. At such alignment points dis96 is able to stop interpreting input bytes as code.

4.1.2 Wrapper Algorithm

The wrapper algorithm follows a simple automation idea of a manual code recovery process. In manual recovery, disassembly starts at an entry function. The recovered code is then read and every call to a function is used to annotate the call destination as a new function. The same process repeats for every jump found in the code, annotating the targets as code blocks of functions. Attention must be paid to stop the interpretation of the code at points where misalignment can occur.

The description so far only covers jumps/calls with explicit addresses. If an indirect jump is found that is not trivial (eg. not a variation of return-from-function), the possible targets must be resolved by reading and understanding the code. This is especially easy if the indirect jump is part of a compiler pattern.

This leads to the following idea of an algorithm. Read the input file and locate the start of a function. Inside of a function, process the input lines and check for direct jumps and calls. Emit new disassembly commands for these calls and labels. Pay attention to stop interpretation if an unconditional jump ends a function. Resolve indirect jumps according to known patterns.

The most important indirect jump that needs resolution is the jump part of a switch idiom. The analysis proceeds in two steps: In the first step of this resolution the analysis marks the jump table and in the second step it reads the possible targets from the jump table.

Statement	# targets	block end
jump	1	Y
call	1	N
return	1	Y
conditional jump	2	N
halt/rst	0	Y

Table 4.2: Control Flow Manipulating Statements Abstract Semantics

```
m = entrypoints
while c changes do
    c = disassembly(m,bin)
    m ∪ = process(c,m)
end while
```

Figure 4.1: cf96 Algorithm Pseudocode

Both the detection of function ends and resolution are problems that cannot be solved generally using syntax. This algorithm makes use of heuristics to answer them.

The algorithm is independent of the actual control-flow manipulating statements[1]. It only uses an abstract semantic of these statements (see table 4.2). It decides between statements that have calculated targets (indirect jump/call), statements with immediate targets (direct jump/call), statements with targets stored on the call stack (return), statements with two successors (conditional jump), and statements with no successors (halt).

Both the heuristics and actual statements are described in section 5.1.2.

Algorithm

The algorithm implements a recursive traversal disassembler. It depends on a linear sweep disassembler for the actual disassembly and heuristics to resolve indirect jumps. Its input is a set of entry points into the control flow graph.

The main output of the algorithm is the input to the linear sweep disassembler. This output gets refined on each loop and finally resembles all information that the sweep disassembler needs to fully disassemble the binary file.

[1] Non control-flow manipulating statements are processed using the semantics of a no-operation instruction.

```
1: for all disassembly lines do
2:     if functionstart then
3:         mark in_function
4:     end if
5:     if in_function then
6:         if direct jump or call then
7:             mark dest
8:         end if
9:         if unconditional jump then
10:            check if function ends
11:        end if
12:        if indirect jump then
13:            if switch idiom then
14:                mark jumptable
15:            end if
16:        end if
17:    end if
18:    if jumptable then
19:        while valid entry do
20:            mark entry
21:        end while
22:    end if
23: end for
```

Figure 4.2: cf96 Statement Processing

The algorithm (fig. 4.1) consist of a loop that triggers disassembly and processes the result file, until the result file does not change. It is initialized with the known program entry points. On each round the result of the processing of the disassembler output is accumulated and used as annotation of the binary in the next round.

Figure 4.2 shows the processing of the disassemblers output to calculate new annotations. The output from the disassembler is processed line-by-line. The interpretation of the input lines provides two functionalities: (1) recover the possible control flow by processing instructions and (2) find possible targets of indirect jumps stored in tables.

Except of the detection of the end of a function and the decision whether an entry of a jump table is considered valid all of the decisions and detections are completely syntactical.

Interpretation of instructions is suspended until the start of a function is detected (cmp lines 2–5). Each function start is given explicitly in the disassembler output by a function header.

Inside of functions the instructions are interpreted according to different cases:

Line 6 Every jump or call marks its destination if it is not already marked.

Line 9 Every unconditional jump/call or no successor instruction is used to check if the function is left.

Line 12 Indirect calls/jumps are checked whether they belong to the switch idiom and processed accordingly.

The jumptable idiom marks the address of its target table as a data location.

Since the disassembler also decodes data locations into the output file, the reconstruction of jumptable targets can be directly included in the processing of statements.

If a previous iteration has marked a memory position as jump table, the table is processed and entries are added as targets if they fulfill the switch idiom heuristic.

The newly marked locations are merged with the already known markers and the overall process is restarted. The merge operation ensures that there is a limit of one code marker and one data marker per location.

Termination

The algorithm performs a breadth-first search (BFS) on the control flow graph. It marks every node it has traversed.

The control flow graph is of finite size, because every instruction is at least one byte long, and the address space and therefore the number of instructions is limited (to 64k bytes or instructions).

The BFS traversal of the finite sized CFG is also of finite size and the algorithm will terminate.

The disassembly operation can be considered as a monotonic operation. If more annotations are input, more output is received. Each time the disassembler output is processed new markers can be calculated by the processing part of the algorithm. The merging of new markers with all previously known ones implicitly forms a power-set lattice in which it is only possible to go upwards.

Since the algorithm never drops knowledge in any of its steps and is bounded by its lattice, it must either reach a fixed point or the top element of its lattice. For both the fixed point or the top element, the disassembler output will stay the same in two loops and the algorithm will terminate.

Runtime Complexity

The worst case complexity of the algorithm is very simple to calculate. The outer loop terminates after at most $\mathcal{O}(n)$ iterations, adding a single marker for every single one out of n addresses at a time. The inner processing of the loop is dominated by the merge operation of the markers, which for a trivial, non-parallel implementation has a complexity of $\mathcal{O}(n^2)$ operations. This leaves in total for the worst case $\mathcal{O}(n^3)$ operations.

It is important to note, that the outer loops number of iterations is controlled by the depth of a BFS traversal of the CFG. In best-case of a flat CFG there would be only three necessary iterations. On average the BFS tree of the CFG's height is related to $\log(n)$.

The inner loops statement processing part of the algorithm processes every line of the disassembler output in a linear manner, resulting in a complexity of $\mathcal{O}(n)$.

The disassembler current implementation sorts its input data (markers), and then linearly processes the input bytes, thereby being dominated with complexity $\mathcal{O}(n^2)$ by its sort implementation.

Correctness

Soundness of a code reconstruction means that if the algorithm finds a statement, it is one in the source. A complete reconstruction means that if there is a statement in the source code, the algorithm finds it.

The algorithm described in this section is neither sound nor complete on general input. It is sound if some assumptions about the input to the algorithm and the idioms used to process the input hold.

The soundness criteria cannot be fulfilled on general input because the algorithm does not contain a data-flow analysis and therefore cannot solve the indirect jump problem for every possible input.

The completeness criteria is impossible to reach on a general input binary, because the control flow in the binary may contain unconnected parts (for example an uncalled function from a library).

To be able to calculate a sound result the following statements must hold:

Input The input binary must contain a control-flow graph without edged that lead out of the control flow. This disallows bogus targets (jumps to positions where no code exists).

Input The code may not break the procedure call standard conventions (e.g. if a function returns, it must return to its caller).

Input/Idiom The functions must not interleave in the binary.

Algorithm The algorithm must approximate the control-flow manipulating statements in a way that they to not introduce unsound edges.

Idiom The idioms that are used to calculate the targets of indirect jumps may not introduce unsound edges[2].

A compiler generated binary without obfuscation, for which approximations for the indirect jumps are available as idioms will hold the above assumptions.

4.1.3 Comparison to Existing

The proposed method of control flow recovery is implemented in a very similar way in the commercial tool IDA pro (see page 32 in [17]). The main difference is given by IDAs different disassembler component. The IDA disassembler is able to disassemble single statements at a time and IDA can therefore implement a depth-first search (DFS), which reduces the necessary work to keep and process the code markers. In this algorithm a BFS traversal is chosen because it fits the model of alternating calls to the sweep disassembler and phases of calculating new positions in the code. In each round the sweep disassembler can uncover the next depth in the control-flow graph. Generally there is no difference in using a DFS or BFS to recursively reconstruct the code.

4.2 Validation

In forward engineering each development step's result is validated against some previously set requirements. Reverse engineering without access to the original specification has the problem that there is no ready-made set of requirements to validate results against.

[2]The switch idiom must be tuned to each input individually.

We discuss different kinds of imprecisions in (reverse-)engineering steps and how to check approximations that needed to be made.

In the following different error sources and possible remedies are explained.

4.2.1 Error Sources

Reverse engineering of a binary into source code is split into two basic phases: the acquisition of the binary image and the disassembling of the code in the image. Both steps can introduce different errors that may propagate into the next steps.

A special problem is the lack of error messages in the individual steps.

Disassembler Each class of disassembler can cope with different input data. Section 1.4 and chapter 3 give an overview about the possibilities of different approaches. The problem is that the individual approaches will not be able to signal if they encounter code that they cannot process without errors.

A second problem is that a disassembler may have a bug in its instruction decoding stage and provide an erroneous interpretation of the bytes.

Image Acquisition Most analyses can safely assume a valid executable image, but some might distrust their image acquisition process.

In embedded systems reverse engineering, the first errors can be introduced during the acquisition of the firmware image to analyze.

The first problem is that the ROM could have bit errors. Such random bit errors can be temporary or permanent. If temporary, they can be recovered easily using multiple readouts of an image.

The second problem is that the readout process might introduce additional bit errors. A common source of these errors is a contacting problem of parallel memory chips.

To further complicate this issue, neither standard ROM circuits nor the readout process have error checking facilities, thereby making it impossible to gain error reports at this stage.

It is easy to mitigate these problems if checksums of the ROM image are available.

4.2.2 Remedies

In the following paragraphs different remedies that address the problems during code reconstruction and help to increase trust in reconstructed solutions.

The following remedies sum up and provide a very solid check of the reverse engineered result.

Tests When addressing the problem of disassembler capabilities, tests using binaries with known source code provide knowledge about what reconstructions are performed.

For many systems it is possible to acquire development tool chains, allowing the reverse engineer to produce executables for the architecture, and thereby providing him with data (additional executables) and a specification (known source code) to validate against. It is important to check, if these executables follow the same patterns as the executable(s) under test, to be able to notice if the tested analysis functionality is actually used in the reverse engineering process.

Testing with known binaries works parallel to the following approaches and assesses the individual tools capabilities.

Tool Output Comparison If it is possible to calculate the output of an analysis' step using tools that follow a significantly different approach a comparison of the tools output is a promising technique to increase trust in the result without knowledge of a specific domain.

If the control flow of an executable, calculated by different approaches is identical, it can be assumed that the individual tools all validate against the same specification.

If a step can be performed in multiple ways it is in general a good way to validate the output of one tool against others.

Sanity Checks Sanity checks are relatively easy to perform checks that can be performed immediately after a result is received to perform a quick check that is supposed to show existing problems.

Despite the lack of a concrete system specification, for many systems it is not feasible to produce software and hardware that breaks the conventions of what is considered usual. An engineer that has knowledge of what is usually built in the domain of the analyzed device can tell if the results of an analysis step look sane.

- In an executable image that communicates with human beings, it is quite common to find ASCII (or some other encoding) strings, containing messages that relate to the system's domain.
- Assembler code recovered from a binary file should usually locally operate on a bounded set of registers.
- If the code calculates results in one register, and then never reads this again, something is usually wrong.

Sanity checks can be performed after any step in the process. Looking for ASCII strings helps to check image acquisition, while checks on the reconstructed code's macro semantics provides good checks about the usefulness of the code.

Selfcontained Validation A very specialized sanity check that can be performed after the actual reconstruction work is completed, is the search for a selfcontained validation.

Code running in safety critical environments can contain functionality to check itself.

An example is the use of checksum functions calculated over the program code. The result of these checksums is compared with known checksums of the same code. If they match, a fairly good assurance about the unchangedness of the program code compared to the original calculation can be given.

If the functionality to check itself is recovered from a program code it is possible to rebuild it in an external software and redo the calculation of the checksums. If they match the ones that are used by the selfcheck the same assurance about unchangedness can be given.

The described remedies provide adequate mitigations for the problems that the core reconstruction process might encounter. Using them each step in the process can be checked and confirmed, usually with more than one individual method.

5 Experimental Validation

To evaluate different code recovery strategies a real world example binary from an embedded control device is analyzed.

The necessary changes to existing tools and platform details of the cf96 tool that instantiate a concrete version of the algorithm described in 4.1.2 are described.

The disassembly results of a test input and the real binaries results are compared. The test input is specially designed using a C compiler for the system and a source file with variations of switch statements. The real binary is acquired from an example control system provided by an industry partner. On both files code reconstruction is performed using the available tools and an assessment is performed whether their results are acceptable.

5.1 Example Control System

The system consists of a microcontroller (MCU) and periphery devices that are accessed via a bus system.

It is sufficient for an analysis to work with an abstract memory interface. We summarize the arrangement of the system's memory in ■. Section 5.1.1 describes the implications of the memory system for the analysis.

Sections 5.1.2 to 5.1.4 gives the MCS-96 platform information that provides semantics and heuristics for the control flow recovery.

5.1.1 Memory Architecture

The practical implication of the memory architecture is the existence of two different entities in different memories but at the same address. Variables are allocated at positions already occupied with code.

It is a common assumption for tool design that one address is uniquely associated with a single code label or a single data label. This assumption prohibits the naming of variables (data access) if a function already exists.

It is a common sanity assumption for compiler generated code to not allow the introduction of additional misaligned labels inside of an instruction.

To solve the issues of a unique mapping between address ranges and instructions, respectively data locations the following two solutions exist.

It is possible to drop the sanity check and use two different symbol stores, one for code and one for data. Using this approach it is possible to have two different entities at the same address.

A less invasive approach is the usage of an offset for every data access. By adding an offset to every data access it is possible to move the data access space into an otherwise unused address region.

To use stack pointer or frame pointer based access heuristics for local variables and parameters of functions it is possible to only add the offset to accesses not using the R18 (stack pointer) or R1A (frame pointer) as index.

5.1.2 Control Flow Manipulating Statements

This section provides a detailed list of control flow manipulating statements of the MCS-96 architecture together with a description of their code recovery influencing semantics.

Conditional Jumps The conditional jump instructions evaluate an expression and either jump to their target or normally continue control flow, by executing their successor.

The "**j**cond dst" instruction evaluates its condition (cond) and either resumes execution with the next statement in memory or sets the program counter to (dst) if it evaluated its condition to true.

The "**jbs**/**jbc** num, reg, dst" instruction tests if the bit (num) is set (s) or clear (c) in the register (reg). On positive evaluation it jumps to its target. The control flow is resumed with the next statement otherwise.

The "**djnz** reg, dst" instruction decrements the register (reg). If the register (reg) is valued 0 the control flow is resumed with the next statement. Any non-zero value executes the jump to (dst).

Unconditional Jump The unconditional jump (**ljmp**/**sjmp** dst) instructions set the program counter to the target given as the (dst) operand. They never automatically execute the next statement and can therefore end a sequence of instructions.

Indirect Jump The indirect branch (**br** reg) instruction behaves like an unconditional jump. The difference is that the target is given in a register (reg) and not as an immediate address. It can also end a sequence of instructions.

Unconditional Call The unconditional call (**lcall**/**scall** dst) instructions push the address of the next instruction in memory on the stack and execute a jump to their target. If the called function returns, the control flow resumes at the next statement.

Return From Function The **ret** return-from-function instruction takes the topmost element (pop) from the stack and sets the program counter to its value. The return function does not resume control flow at the next instruction in memory and can therefore end statement sequences.

Special The **rst** reset instruction triggers a hardware reset of the process. The control flow will resume at the reset address `0x2080`. It never executes the next statement in memory and can therefore end statement sequences.

5.1.3 Program Entry Points

The entry points of the control flow graph on the MCS-96 architecture stem from the reset location and from the interrupt tables of the processor.

Reset The processor needs to start execution at an address after it is powered on. This address is called the reset address. It is the initial value of the program counter after reset. Usually the startup code of a compiled software inserts a jump instruction to the main function. The reset address of the MCS-96 architecture is `0x2080`.

Interrupt Vector The interrupt vector stores the addresses of interrupt service routines. It is used as a table to store the locations of functions that are called by the hardware to handle an interrupt.

There are special interrupt locations that do not call their target but directly execute a jump, which prohibits normal resumption of control flow. Such interrupts are NMI (non maskable interrupt) or unimplemented opcode interrupt.

Irregardless of the execution of call or jump semantics, it is assumed that code exists at the interrupt locations.

Two interrupt vectors of length eight exist on the MCS-96 architecture at addresses `0x2000` and `0x2030`. The locations of the trap and unimplemented opcode interrupts are `0x2010` and `0x2012` respectively.

```
        ld reg1, jump_table[reg2]
        br reg1
jump_table:
        dw target1, target2, ...
```

Figure 5.1: MCS-96 Switch Idiom

5.1.4 Compiler Idioms

This section introduces two idioms to resolve indirect jumps. Indirect jumps on the MCS-96 can be spotted easily as there exists only one[1] indirect jump instruction: **br**.

The idioms are discovered by an exhaustive search for indirect branches in the system's software. We discuss three different idioms, two of which are concerned with the resolution of indirect jumps and one that provides knowledge of the layout of procedures in memory.

Switch Idiom

Compilers frequently make use of indirect jumps in the implementation of switch idioms (see section 2.1.2).

The MCS-96 switch idiom consists of two instructions and one data location (see fig. 5.1). The first instruction **ld** loads an entry from the table (jump_table). It usually uses the same register as index of the fetch and destination for the fetched entry. The second instruction is **br**. It uses the fetched target and jumps to it.

The existence of a switch idiom with usage of jump tables is confirmed in section 4-41 of [23].

To detect the switch idiom it is sufficient to match on the instruction sequence **ld** reg, loc[reg], **br** reg. If both operate on the same register, it is safe to assume a switch idiom and mark the memory location as a jump table.

To resolve the target of a switch idiom jump the entries from the jump table must be available to the analysis. It is trivial to perform an index fetch for such an entry, but the index is unknown. If one assumes that the table contains a compact list of entries at the beginning, the problem is reduced to developing a heuristic to detect the end of such a list.

[1] The abuse of return as indirect jump is excluded.

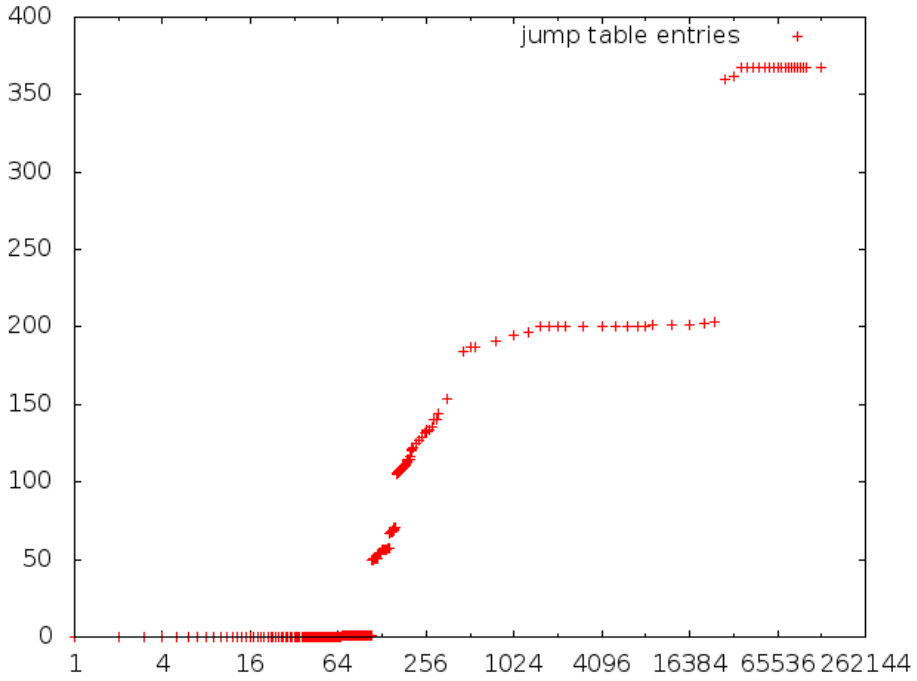

Figure 5.2: Maximum Entry Distance to Function Entry Point vs. Recovered Entries

Assuming the locality of jumps in the code (see [8, 3]) it is possible to provide a very simple heuristic the detect the end of the table. Every[2] table entry must be within a certain arithmetic interval to the function entry point. The first entry that is not within limits marks the end of the table.

To recover the entries of the jump tables of the binary in this work an arithmetic interval of ± 4096 bytes to the function entry point was sufficient.

Figure 5.2 shows the variation of the heuristics parameter. Using the example system software, code reconstruction is performed with the heuristic parameter and the total number of recovered table entries is plotted. For small values, no entries are recovered as there usually is some code between the function entry and the position of the jump table targets. Between 64 and 2048 the number of recovered entries increases significantly. Beyond 16384, another increase can be noted. These are spurious entries. The num-

[2]This definition allows empty tables.

```
            add R1C, R1C
            ld R1C, 0x7123[R1C]
            br R1C
0x7123:
            dw 0x7150, 0x7180
0x7150:
            op1
            op2
            ret
0x7180:
            op3
            op4
            ret
```

Figure 5.3: MCS-96 Switch Idiom

ber of (spurious) recovered entries per table is implicitly limited by other table and data references.

The first increase in values correlates with the valid recovered table entries. The second increase happens when invalid entries are added to the switch idioms targets.

Figure 5.3 shows an example instantiation of the switch idiom used by the Intel MCS-96 compiler. The starting addresses of the case blocks are stored in a table at `0x7123`. The **add** instruction is used to double the switched on variable and make it usable as table index. The **ld** instruction then fetches the target address into R1C and the **br** instruction jumps on R1C, leading the control flow to execute the case blocks.

Optimized Return

The second usage of the **br** instruction is in a pattern that the compiler uses to return from a function, while simultaneously freeing local variables and parameters on the stack.

In the example system's executable all of these **br** instructions are performed on register R22.

Figure 5.4 shows the optimized return pattern. The first instruction loads the return address from the stack. The second instruction frees, by adding a value to the stack pointer, corresponding to the size of the freed variables,

```
ld      R22, cnt1[R18]
add     R18, cnt2
br      [R22]
```

Figure 5.4: MCS-96 Optimized Return Idiom

the remaining location on the stack. The third instruction performs the jump the return address.

Detection of this pattern in this binary is extremely easy because the compiler uses fixed registers to perform this operation. A simple detection of **br** [R22] detects all occurrences.

Function Layout

To decide whether or not a function ends at a given memory address, a simple model of the function layout in memory is used. A function is assumed to have a unique lowest and highest address forming an interval that is disjunct to every other functions interval. The decision is a simple check of the interval bounds.

This function model only works if the compiler/linker does not use inter-function block reordering to perform optimizations.

5.2 Tool Adoptions

From the available tools IDA Pro is selected to provide a compare to recursive traversal disassembly of the executable. Jakstab is the only tool to use a data-flow sensitive code reconstruction. Both available tools are customized to support the MCS-96 controller and specialized memory architecture of the system.

5.2.1 IDA Pro

IDA Pro supports the MCS-96 architecture in principle. The drawback of IDA's support is the missing ability to redirect data accesses into a memory area separated from the code area.

The dual symbol-store approach (see section 5.1.1) is impossible to implement in IDA. The symbol store of IDA is not customizable, which makes it impossible to separate the storage of data and code symbols. This is needed to support data and instructions that occupy the same memory.

The second approach (see section 5.1.1) to support the system's memory architecture is the implementation of an offset feature that allows the addition of a numeric offset on every data access. This implementation is possible because the IDA software development kit (SDK) provides the source code for the MCS-96 architecture module.

Because of problems compiling the CPU module from the IDA SDK, a reimplementation of the CPU module using IDA's python API was necessary.

Implementing a disassembler CPU module for IDA is very simple. Three functions need to be implemented. The `ana` function reads bytes at the current file position and decodes them into one instruction. The `out` function processes the datastructure of an instruction and prints a textual representation of it. The `emu` function analyzes an instruction concerning its impact on the control flow and data access. It provides information about all the possible next statements of an instruction and all of its data accesses. This information could include indirect accesses, but these usually cannot be resolved at this stage.

IDA does not have a switch idiom heuristic for the MCS-96 architecture and each of the occurrences has to be handled manually using the switch idiom wizard. The wizard take the position of the indirect jump and the length and position of the jump table and introduces control-flow edges from the jump to all targets.

5.2.2 jakstab Adoptions

Jakstab so far only supports the x86 architecture. To use it on the MCS-96 binary, it must be extended.

To add a new architecture to the jakstab static analyzer it is necessary to implement a disassembler, extend the assembly language classes to be able to hold the disassembled code, provide a SSL file of the central processing unit (CPU) and extend the intermediate language to support special semantics of the hardware.

Jakstab uses the class hierarchy from the OpenJDK HotSpot-Serviceability-Agent as base classes for its assembly language representation ([16]). A new architecture needs to subclass this hierarchy and provide implementations matching its specific instruction syntax and operand semantics.

Jakstab requires a disassembler class that implements one function to disassemble one instruction at a time for a given memory address. There is no additional disassembler infrastructure so the MCS-96 disassembler is imple-

mented as a opcode-lookup-table based disassembler using parameterized decoding classes to provide instruction-group-based disassembly.

The MCS-96 SSL file is based on the instruction semantics given in table 3-1 in [13]. This table provides a register-transfer-like description for the MCS-96 instructions.

The MCS-96 register architecture provided in the SSL file lists all the available general purpose byte, word, and double-word registers and defines the shared bit-ranges between them.

We developed a assembler class hierarchy to hold the statements, a disassembler, the SSL file, and adopted the assembler to intermediate language mapper.

5.3 Results

This section describes the experiments inputs, the methods used to compare the tools output, and examines the comparison.

5.3.1 Test Input File

To test the tools with a smaller input file, containing a known control-flow graph, a sample C code is developed. With this file a first sanity check about the general ability of the tools is possible. Since the file also contains the important switch idiom it also serves as test for the general ability of the tools to resolve it.

The C code contains several functions that use switch as their control structure. It is designed to contain a data dependency from one function to another selecting a single case statement out of a large set of cases.

The test binary for the disassemblers is compiled from C code using a compiler similar to the compiler of the original system.

The switch statement in the C code with 40 cases triggers the use of the jump table idiom in the compiler.

In summary the test file contains a loop-free callgraph between several functions. Between the functions exists a data dependency that allows dead-code elimination in one function. One switch statement is implemented using a jump table.

5.3.2 Experiment Settings

The experiment compares the disassembly results of two recursive traversal (cf96 and IDA Pro) and one iterative (jakstab) disassembler. The experiment helps to establish trust in the results of the tools disassembly output.

All of the tools need entry points to perform their work. Jakstab has a configurable abstract domain that plays a role in the precision of its results and is therefore also part of the experiment settings.

The recursive traversal disassemblers with disabled support for the switch idiom are expected to be unable to recover the complete control flow. With enabled switch support, they are expected to recover all possibly reachable code. The iterative data-flow sensitive approach is expected to perform an implicit dead code analysis and provide a smaller (compared to the complete possibly reachable) but more precise (stripped from unreachable code) result.

Entry Points None of the tools in this experiment is able to heuristically discover the program entry points. Each tool is given the set of program entry points as initial addresses for the code recovery.

The microcontroller manual ([13]) defines the possible entry points. The reset location of the architecture is fixed and always the same at `0x2080`. The other entry points are saved in tables in the executable itself.

Table 5.1 lists all program entry points of the analyzed system. Three different kinds of entries can be distinguished. At the address of the reset location, the binary contains code (jmp code) that jumps to the main function. The interrupt tables at 0x2000 and 0x2030 contain addresses that the controller hardware calls (int call), if the corresponding interrupt occurs. The interrupts at 0x2010 and 0x2012 are not called but jumped to (isr jump).

The test file does not contain interrupt routines. This makes the reset location address sufficient to discover the complete control flow.

Jakstab The abstract domains in jakstab have an influence of the precision that is achievable. Two abstract domains from the jakstab framework are used to analyze the binaries.

The constant propagation domain is designed to provide a simple and fast disassembly. It is not able to model a call stack, but jakstab contains a so called optimistic resolver for function calls and returns that allows the constant propagation domain to perform a whole program analysis.

Location	INT	Name	Kind	Target
2080		reset location	jmp code	[redacted]
2000	00	Timer overflow	int call	[redacted]
2002	01	AD Conversion Complete	int call	[redacted]
2004	02	HSI Data	int call	[redacted]
2006	03	HSO	int call	[redacted]
2008	04	HSI0 Pin	int call	[redacted]
200A	05	SW Timer	int call	[redacted]
200C	06	Serial	int call	[redacted]
200E	07	External Int	int call	[redacted]
2010		Trap	isr jump	[redacted]
2012		Unknown Opcode	isr jump	[redacted]
2030	08	Serial TI	int call	[redacted]
2032	09	Serial RI	int call	[redacted]
2034	10	HSI FIFO Half	int call	[redacted]
2036	11	Timer2 Capture	int call	[redacted]
2038	12	Timer3 Overflow	int call	[redacted]
203A	13	External Int1	int call	[redacted]
203C	14	HSI FIFO Full	int call	[redacted]
203E	15	Non Maskable Interrupt	int call	[redacted]

Table 5.1: MCS-96 Entry points. (The shown target is extracted from the given firmware image)

Bounded address tracking (see section 3.3.2) is the default domain. It is capable to perform a precise analysis providing its own model of the call stack.

5.3.3 How to Compare

The analyses provide output in the form of text files containing lines with pairs of addresses and instructions. The output is annotated with control flow information between the instructions.

IDA and cf96 provide control flow information using labels to annotate the destinations of jumps and calls. In the output no information on calculated targets of jumps and calls is provided at the caller/jumper side.

Jakstab does not annotate its output with labels. It instead provides information about to and from addresses at every statement where these are not the previous or next statement.

Compare CFGs

From the assembly text recovered by the tools a control flow graph can be calculated and compared.

A control flow graph is consists of a set of basic blocks and a set of connections (pairs) of basic block exists and corresponding entries.

A basic block is built from a linear (one entry, one exit) sequence of statements that are executed one after the other. The entry may be jumped/-called to from more than one block and the exit may jump to more than one other block. Basic blocks can contain calls to function.

Maximal basic blocks are basic blocks that are formed by concatenation of basic blocks, where the first basic block has only one jump target at its exit and the second block is only jumped to from the first. It an analysis does not provide explicit information about single exits and single entries of basic blocks it is impossible to perform such a concatenation that are not consecutive in memory, resulting in non-maximum basic blocks.

These graphs could be compared for common subgraphs, but a much simpler comparison of the output is possible. This simplification does not weaken the comparisons results concerning the overall reconstructed code and transforms the problem from a computationally expensive (common subgraph search) to a simple linear line by line comparison.

Simplification The goal of the analysis is the discovery of reachable code, meaning the information how the control flow to this code actually looks is

not relevant for the comparison of the code recovery. Therefore the output comparison can be simplified to just compare the existence of the same instruction at the same address.

A second simplification is to compare only the operators of the instruction. The tools use a slightly different syntax to print the operands of an instruction. By comparing only the opcode and not the complete instruction a loss of information may be introduced but its scope can be limited.

If two tools identify the same opcode at a given address but different operands, this error can only be introduced by a faulty operand decoding in one of the tools.

If the instruction decoding part of the tools is not suspected to introduce errors, the existence of the same operators at identical addresses is sufficient to assume identical decoding.

5.3.4 Tables

This section describes the output of the tools for the test code and the example system's code.

The previously described test input file is processed by all three tools and is compared on a basic block level.

The systems binary image is processed using the same tools but due to the size of the output and cost of a manual basic block analysis, it is compared on the instruction level.

Test Code

The code recovery of three tools in a sum of five configurations was used to analyze the file containing the generated test code.

Jakstab runs on the file with bounded address tracking as abstract domain for the configurable program analysis. cf96 processed the file with the heuristic for switch idiom recovery on and off. The output of IDA Pro is extracted before (without switch) and after the manual recovery of the switch idiom.

Table 5.2 lists information about the recovered code. For each tool, the number of function entry points, the lines of code, jump and call instructions, indirect jumps and the number of basic blocks in the code is listed.

The results of the tools can be grouped by the number of lines of code recovered. This grouping shows the correspondence of the results of cf96 and IDA Pro with and without switch idiom reconstruction. Both tools recover identical lines for the same configuration (109 lines or 189 lines).

Tool	# Fkt	# LOC	# jumps	# calls	# indirect j/c	# bblocks
cf96	6	109	19	9	1	29
IDA Pro	6	109	19	9	1	23
jakstab	6	111	20	9	1	24
cf96 + sw	6	189	59	9	1	69
IDA Pro + sw	6	189	59	9	1	63

Table 5.2: Code Comparison: Test Code

The difference stems from the 80 lines of code in the switch idioms case statements.

Jakstab, with 111 lines, reconstructs only one of the case statements (2 lines).

A minor difference in the output of the tools exists in the number of recovered basic blocks. Table 5.3 shows the number of maximal basic blocks calculated from each of the outputs. The same non maximal basic blocks exist in the output of cf96 and IDA Pro. With the output of IDA Pro it is possible to append 6 blocks to their predecessors. While these non-maximal basic blocks show a possible imprecision they are of no influence to the code reconstruction as it does not influence the number of reconstructed statements (lines of code).

The configurations of cf96 and IDA Pro recovered 80 lines of code (or 40 basic blocks more) if the switch idiom support was used, corresponding exactly to the 40 switch cases in the test file C code.

Jakstab with its bounded address tracking domain was able to calculate the 23 maximum basic blocks as well as one case of the switch idiom. The code in the test file implementation sets the value to be switched on to this specific case. Jakstab can determine this fact using constant-folding like behavior included in bounded address tracking. Both cf96 and IDA Pro are not able to determine this fact, because they do not perform data-flow analyses and cannot propagate the knowledge of a single possible value for a variable from one function to another.

	jakstab	cf96 + sw	cf96	ida + sw	ida
jakstab	23+1=24	24(=) 45(+) 0(-)	24(=) 6(+) 1(-)	24(=) 39(+) 0(-)	23(=) 0(+) 1(-)
cf96 + sw	24(=) 0(+) 45(-)	23+6+40 = 69	29(=) 0(+) 40(-)	63(=) 0(+) 6(-)	23(=) 0(+) 46(-)
cf96	24(=) 1(+) 6(-)	29(=) 40(+) 0(-)	23+6=29	29(=) 40(+) 6(-)	23(=) 0(+) 6(-)
ida + sw	24(=) 0(+) 39(-)	63(=) 6(+) 0(-)	29(=) 6(+) 40(-)	23+40=63	23(=) 0(+) 40(-)
ida	23(=) 1(+) 0(-)	23(=) 46(+) 0(-)	23(=) 6(+) 0(-)	23(=) 40(+) 0(-)	23

Table 5.3: Basic Block Comparison: Test Software, reset entry point

Controller Code

The code reconstruction for the system's code uses the same tools with similar (jakstab) and identical (cf96, IDA) configurations, with the addition of the input of 1 or [redacted] entry points. Limiting the control flow by setting only a single entry point helps to keep the control flow small enough to manually make observation.

[redacted] shows the metrics calculated for the output of the tools. The lines are grouped using the number of recovered lines of code.

With the input of one entry point into the tools, all three tools were able to recover [redacted] functions with [redacted] lines of code. The reconstruction of the reachable code stopped at one indirect jump of the switch idiom.

With enabled support of the switch idiom, IDA and cf96 recover the complete reachable code from the entry point and process two switch idioms. Jakstab is not able to resolve the data-dependent jump in this case.

Jakstab provided identical code reconstructions for bounded address tracking and constant folding with optimistic function resolution. Without optimistic function resolution there are no realistic results (less than [redacted] lines of code).

Using all twelve available entry points to the code the data shows a similar result as with one entry point.

Jakstab, together with the tools with disabled switch idiom support, recover [redacted] lines of code in [redacted] functions.

The resolution of the switch idioms allows IDA and cf96 to further process the file and recover [redacted] lines of code in [redacted] functions using a total of [redacted] recovered switch idioms.

The analysis of identical instructions at the same addresses shows a growth only relation between the results (see [redacted]). The four result groups (switch/noswitch, 1ep/[redacted]ep) can be ordered by the number of lines recovered. Each result is completely contained in the next result. This result shows that although some of the tools do not recover the whole possible control flow, the recovered code is part of the complete result.

5.3.5 Summary

The code reconstruction using the test code shows the expected results for all of the tools. Without switch support the recursive descend approaches are not able to recover the whole reachable code. Using an idiom recovery allows reconstruction of the whole program. Jakstab is able to use its abstract domain and perform a code reconstruction with an integrated dead code analysis, limiting the number of case statements to the single one that would be executed on real hardware.

The code reconstruction for the system works as expected with the exception of an incomplete reconstruction using jakstab. The lightweight approach is capable to reconstruct the code with the same coverage and accuracy as the state of the art IDA Pro disassembler. Jakstab shows with the test file that can reconstruct switch behavior precisely but either the MCS-96 addition or the abstract domain configuration prohibited a calculation of the possible switch targets contained in the example system's image.

6 Conclusions

Code Recovery IDA Pro and cf96 produce identical results. The cf96 approach of implementing the bare minimal semantics of the assembly language and using compiler idioms to analyze the code only when needed provides a simple solution that competitive with the industrial standard solution of code recovery.

Generally, it seems to be possible to solve the code reconstruction with simple tools if the stack and function call semantics of the code hold and indirect jump operations are approximated by heuristics.

While jakstab is capable of precise control flow reconstruction for the test file, it could not recover the whole code of a real world problem. It seems improbable that the analysis engine itself is the source of the problem, leaving the statement and memory abstractions as well as the abstract domain configuration as source.

Further Work Providing answers to the industry questions about the high-level concepts in the program needs annotations of functionality clusters in the source code. Creating such annotations manually is very time consuming. Automated analyses of shared (indirectly addressed) variables between functions and propagation of names along such def/use paths could speed up this process.

While it is possible to recover all executed statements of an executed program using simplified execution instructions, automated code splicing and sophisticated data flow analyses (points to, aliasing, ...) need precise semantics of the statements to preform sound operations. Implementing single tools with precise semantics per architecture seems forbidding. The approach of jakstab to translate the assembly instructions to an intermediate language and perform its analyses on this language seems promising.

The binary analysis platform (BAP, [2]) follows this approach but so far only provides a very simple (linear sweep) tool (`toil`) to transform executables with symbol information of the x86 platform. It looks very promising to build an intermediate language lifter using recursive descend and heuristic support disassembly to extend BAP with additional architectures.

Bibliography

[1] Gogul Balakrishnan and Thomas W. Reps. „Analyzing Memory Accesses in x86 Executables“. In: *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Evelyn Duesterwald. Vol. 2985. Lecture Notes in Computer Science. Springer, 2004, pp. 5–23. ISBN: 3-540-21297-3.

[2] *BAP: The Next-Generation Binary Analysis Platform*. URL: `http://bap.ece.cmu.edu/`.

[3] Laszlo A. Belady. „A Study of Replacement Algorithms for a Virtual-Storage Computer“. In: *IBM Systems Journal* 5.2 (June 1966), pp. 78–101. ISSN: 0018-8670. DOI: `10.1147/sj.52.0078`. URL: `http://dx.doi.org/10.1147/sj.52.0078`.

[4] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. „Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis“. In: *Conference on Computer Aided Verification*. CAV. Springer, 2007, pp. 504–518.

[5] *Boomerang. A general, open source, retargetable decompiler of machine code programs*. URL: `http://boomerang.sourceforge.net/`.

[6] Cristina Cifuentes and Mike Van Emmerik. „Recovery of Jump Table Case Statements from Binary Code“. In: *Science of Computer Programming*. Vol. 40. 2-3. 2001, pp. 171–188.

[7] Cristina Cifuentes and Shane Sendall. „Specifying the Semantics of Machine Instructions“. In: *Proceedings of the 6th International Workshop on Program Comprehension*. IWPC '98. Washington, DC, USA: IEEE Computer Society, 1998. ISBN: 0-8186-8560-3. URL: `http://dl.acm.org/citation.cfm?id=580914.858217`.

[8] Peter J. Denning. „The Working Set Model for Program Behavior“. In: *Communications of the ACM* 11.5 (May 1968), pp. 323–333. ISSN: 0001-0782. DOI: `10.1145/363095.363141`. URL: `http://doi.acm.org/10.1145/363095.363141`.

[9] Michael James Van Emmerik. „Static Single Assignment for Decompilation“. PhD thesis. University of Queensland, 2007.

[10] *GNU Binutils*. URL: http://www.gnu.org/software/binutils/.

[11] Willem Jan Hengeveld. URL: http://itsme.home.xs4all.nl/projects/disassemblers/.

[12] *IDA: Interactive Disassembler*. URL: http://www.hex-rays.com/products/ida/index.shtml.

[13] Intel. *80C196KB User's Guide*. November 1990.

[14] *Jakstab*. URL: http://www.jakstab.org/.

[15] Neil Johnson. URL: http://www.milton.arachsys.com/nj71/index.php?menu=2&submenu=4&subsubmenu=2&page=7.

[16] Johannes Kinder. Personal Communication. June 2012.

[17] Johannes Kinder. „Static Analysis of x86 Executables“. PhD thesis. Technische Universität Darmstadt, Nov. 2010. URL: http://tuprints.ulb.tu-darmstadt.de/2338/.

[18] Johannes Kinder, Florian Zuleger, and Helmut Veith. „An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries“. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Neil Jones and Markus Müller-Olm. Vol. 5403. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 214–228. ISBN: 978-3-540-93899-6. URL: http://dx.doi.org/10.1007/978-3-540-93900-9_19.

[19] Cullen Linn and Saumya Debray. „Obfuscation of executable code to improve resistance to static disassembly“. In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS '03. New York, NY, USA: ACM, 2003, pp. 290–299. DOI: http://doi.acm.org/10.1145/948109.948149. URL: http://dx.doi.org/http://doi.acm.org/10.1145/948109.948149.

[20] Aleph One. „Smashing the Stack For Fun And Profit“. In: *Phrack* 7.49 (Nov. 8, 1996). URL: http://phrack.com/issues.html?issue=49\&\#38;id=14\#article.

[21] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. „Learning to Analyze Binary Computer Code“. In: *Proceedings of the 23rd National Conference on Artificial Intelligence*. Vol. 2. AAAI'08. Chicago, Illinois: AAAI Press, 2008, pp. 798–804. ISBN: 978-1-57735-368-3. URL: http://dl.acm.org/citation.cfm?id=1620163.1620196.

[22] RTCA. *Software Considerations in Airborne Systems and Equipment Certification*. Tech. rep. DO-178B. 1992.

[23] Tasking. *80C196v6.1 C COMPILER USER'S GUIDE*. 1999.

[24] Mark Weiser. „Program Slicing“. In: *IEEE Transactions on Software Engineering* SE-10.4 (July 1984), pp. 352–357. ISSN: 0098-5589. DOI: 10.1109/TSE.1984.5010248.