Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)

Static Analysis of x86 Executables

Statische Analyse von Programmen in x86 Maschinensprache

Dipl.-Inf. Johannes Kinder
geb. in München

Eingereicht am 24. September 2010

Referent:	Prof. Dr. Helmut Veith
Korreferentin:	Prof. Dr.-Ing. Mira Mezini
Prüfungstermin:	17. November 2010

Fachbereich Informatik
Technische Universität Darmstadt
Darmstadt – 2010 – D17

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 24. September 2010

Johannes Kinder

Acknowledgments

First and foremost, I would like to thank my advisor, Helmut Veith, for his continuing support and his valuable guidance in all aspects of academic life. He gave me considerable freedom in developing my own research agenda and always trusted in my abilities. His uncomplicated way of leading our group allowed everyone to do their best and made it easy to focus on research and teaching without unnecessary overhead.

Furthermore, I thank my colleagues for fruitful scientific discussions and the cheerful hours both on and off campus. Even in tough times, there was always an exceptional spirit of companionship and mutual support. In particular, I would like to thank Florian Zuleger for his contributions to our work on control flow reconstruction; Andreas Holzer for our frequent discussions about the CPA framework; him, Visar Januzaj, and the untiring Michael Tautschnig for proofreading on short notice.

Finally, I want to thank my parents Susanne and Helmut Kinder for their support and their firm belief in me, and Anne-Sophie Dörnbrack for being my emotional stronghold in the never-ending series of highs and lows that is graduate research.

Darmstadt, November 2010

Johannes Kinder

Contents

List of Figures

List of Tables

Abstract

This dissertation is concerned with static analysis of binary executables in a theoretically well-founded, sound, yet practical way. The major challenge is the reconstruction of a correct control flow graph in presence of indirect jumps, pointer arithmetic, and untyped variables.

While static program analysis for proving safety properties or finding bugs usually targets source code, in many potential analysis scenarios only a binary is available. For instance, intellectual property issues can prevent source code from being accessible to verification specialists, and some analyses, such as malware detection, are by definition required to work with executables. Moreover, binary analysis can be useful even in situations where the source code is available, e.g., when the compiler is not part of the trusted computing base.

In most of the existing work, a heuristic disassembler makes a best effort attempt to generate a plain text listing of the assembly instructions in the executable and feeds it to a separate static analysis component. The heuristics render this technique inherently unsound, and the control flow graphs retrieved from such listings are usually fragmented and incomplete. Several approaches have pointed out the possibility of using results of data flow analysis to augment disassembly and control flow reconstruction, but described this connection as suffering from a "chicken and egg" problem, since data flow analysis requires a control flow graph to work on.

This dissertation argues for the integration of disassembly, control flow reconstruction, and static analysis in a unified process. It introduces a framework for simultaneous control and data flow analysis on low level binary code, which overcomes the "chicken and egg" problem and is proven to yield the most precise control flow graph with respect to the precision of the data flow domain. A very precise domain that lends itself well to control flow reconstruction is introduced in Bounded Address Tracking, a combined pointer and value analysis that supports pointer arithmetic. It tracks variable valuations up to a tunable bound on the number of values per variable per program location. Its path sensitivity generally allows strong updates to memory, i.e., heap regions are uniquely identified, and equips it with context sensitivity without assuming a correct layout of procedures.

These building blocks are combined into an extensible program analysis architecture, which is implemented in a novel binary analysis tool. The tool, named *Jakstab*, works directly on binaries and disassembles instructions on demand while exploring the program's state space, allowing it to handle low level features such as overlapping instructions, which cause difficulties for regular disassemblers. The architecture is highly configurable to allow a wide range of analyses, from sound abstract interpretation to heuristics-supported disassembly. Its practical feasibility and improvements over existing approaches are shown through case studies on device driver binaries and system executables found on a regular desktop PC.

Zusammenfassung

Die vorliegende Arbeit befasst sich mit dem Problem der theoretisch fundierten, korrekten, aber dennoch praktisch nutzbaren statischen Analyse von ausführbaren Programmen im Binärformat. Die größte Herausforderung ist dabei die Rekonstruktion eines Kontrollflussgraphen angesichts von indirekten Sprüngen, Zeigerarithmetik und untypisierten Variablen.

Statische Programmanalyse zum Beweis von Sicherheitseigenschaften oder zum Entdecken von Fehlern zielt normalerweise auf Quelltext ab, in vielen potentiellen Analyseszenarien ist jedoch nur eine Binärdatei verfügbar. So kann zum Beispiel die Sorge um geistiges Eigentum verhindern, dass ein Programm Spezialisten zur Verifikation vorgelegt wird, und bei Analysen wie der Erkennung von Schadprogrammen ist grundsätzlich nur eine Binärdatei verfügbar. Darüber hinaus kann eine Analyse von Binärprogrammen aber auch dann Vorteile bringen, wenn Quelltext vorliegt, zum Beispiel dadurch, dass die Korrektheit des Übersetzers nicht länger angenommen werden muss.

In bisherigen Arbeiten zur statischen Analyse von Binärprogrammen wird üblicherweise auf einen eigenständigen heuristischen „Disassembler" zurückgegriffen. Dieser versucht, möglichst alle Assembler-Instruktionen in der ausführbaren Binärdatei im Klartext aufzulisten und gibt diesen Programmtext dann an eine separate Komponente zur statischen Analyse weiter. Die Verwendung von Heuristiken verhindert die Korrektheit dieser Technik, und Kontrollflussgraphen, die aus solchen Programmtexten erzeugt werden, sind meist fragmentiert und unvollständig. In der Literatur wurde bereits von mehreren Autoren darauf hingewiesen, dass die Ergebnisse einer Datenflussanalyse bei der Erzeugung des Kontrollflussgraphen helfen können. Allerdings beschrieben sie diese

Verbindung als ein „Henne-Ei-Problem“, da eine klassische Datenflussanalyse bereits einen Kontrollflussgraphen als Eingabe benötigt.

In der vorliegenden Dissertation wird argumentiert, dass Disassemblierung, Rekonstruktion des Kontrollflussgraphen und statische Analyse in einem einheitlichen Prozess durchgeführt werden sollten. Es wird ein Rahmen für gleichzeitige Kontroll- und Datenflussanalyse auf Maschinensprache vorgestellt, der das „Henne-Ei-Problem“ auflöst und bewiesenermaßen den bezüglich der Genauigkeit der Datenflussanalyse bestmöglichen Kontrollflussgraphen rekonstruiert. Mit „Bounded Address Tracking“ wird eine hochpräzise Analyse eingeführt, die sich besonders gut für diese Aufgabe eignet. Diese Analyse verfolgt sowohl Zeiger als auch Zahlenwerte und unterstützt dabei Zeigerarithmetik. Sie erfasst den Zustand von Variablen bis zu einer konfigurierbaren Schranke für die maximale Anzahl an Werten pro Variable und Programmpunkt. Pfadsensitivität verleiht der Analyse Kontextsensitivität auch ohne eine korrekte prozedurale Struktur annehmen zu müssen, und erlaubt ihr, das Ziel jedes Speicherzugriffs eindeutig zu identifizieren.

Diese Komponenten werden zu einer erweiterbaren Architektur zusammengesetzt, die in dem neu entwickelten Analysewerkzeug *Jakstab* implementiert ist. Jakstab arbeitet direkt auf Binärdateien; während es den Zustandsraum des Zielprogramms durchsucht, disassembliert es bei Bedarf immer nur jeweils eine einzelne Instruktion. Dies erlaubt Jakstab, auch Konstrukte wie sich überlappende Instruktionen zu unterstützen, die herkömmlichen Disassemblern Probleme bereiten. Die Architektur ist sehr fein konfigurierbar, um ein weites Spektrum an Analysen zu ermöglichen, von Abstrakter Interpretation bis hin zu heuristischem Disassemblieren. Der praktische Nutzen und die Verbesserungen gegenüber früheren Ansätzen werden in Fallstudien über Gerätetreiber und Programmdateien eines gewöhnlichen Arbeitsplatzrechners gezeigt.

Chapter 1

Introduction

Reasoning about programs is a cornerstone of computer science. We look at programs to understand whether they are correct or contain bugs, to find out after what time they terminate, or to see whether they conform to our security requirements. Static analysis [109], model checking [39, 114], and abstract interpretation [45] are successful concepts for the formal analysis of programs and have been instantiated in many tools and processes that improve the quality of today's software [12, 16, 19, 20, 52, 75, 103, 137]. At the time a piece of software is written, such tools can be applied to the source code with relative ease. Once the software is compiled into binary format and shipped, however, users further down the line have to trust the vendor and the distributors about the quality and security of the product. This is not only a problem for end-users, but even more so for modular architectures with plugins or drivers, where external companies provide binaries to directly interface with existing software.

Static program analysis, the concept of approximating the semantics of a program to prove or refute properties, is usually targeted at human readable source code written in high level languages instead of low level machine code. The advantages of this common approach are obvious: Source code is easily accessible through text parsing; high level concepts such as loops, procedures, or classes provide a natural partitioning of programs into functionally related units. Yet, there are several compelling reasons to move the analysis behind the compilation process, down to the level of the fully compiled and linked binary. Most importantly, if the analysis targets stripped binary executables, i.e., binaries without symbol or debugging information, it gains the ability to analyze software without access to source code. This ability comes at a price, however, which is the reason why static binary analysis lags behind the development of static analysis on source code. Binaries lack several comfortable features of high level programming languages, such as clearly defined procedures or a distinction between code and data. Absence of symbol information means that variables are not easily identified, but are represented by reusable registers and the memory, which is addressable as a large continuous array. Registers and memory carry no type information, and pointers of any type are indistinguishable from integers.

This dissertation will show how to design a sound static analysis framework that overcomes the difficulties of working with binary executables and low level code. Based on the concept of abstract interpretation, it formalizes the combination of data flow analysis and control flow reconstruction for low level imperative code. It introduces *Bounded Address Tracking*, which allows to analyze binaries at the high precision required for recovering accurate control and data flow information in presence of indirect branches and untyped variables. The framework is implemented in a novel binary analysis tool called *Jakstab* (**Ja**va tool**k**it for **sta**tic analysis of **b**inaries), which allows to combine different analysis components and to trade off precision against coverage of disassembled instructions. An extensive study of experimental results from analyzing real world code demonstrates the practicability and usefulness of the approach.

1.1 Benefits of Binary Analysis

Static analysis of binaries is difficult. From a theoretical viewpoint, the absence of types and structure means that much of the original information present in source code is lost and cannot be used for the analysis. From a practical viewpoint, a great amount of technical detail has to be dealt with diligently, such as dynamic linking, function pointers, or the large number of specialized instructions. Still, the required theoretical and engineering effort is a worthwhile investment for several reasons and opens up multiple avenues of application, which will be outlined in this section.

1.1.1 Alternative to Source Code Analysis

Working with binaries has several advantages over source code analysis, which can motivate an analysis of machine code even when source code is available.

Compiler Independence. Confining the static analysis to source code moves the compiler into the trusted computing base, i.e., any proof over the source code of a program only applies to the final compiled program under the assumption that the compiler provides a fully correct translation which does not modify program semantics. Compilers generally do a good job of preserving semantics, but they do contain bugs, and aggressive optimizations may change the behavior of a program in an unexpected way. For instance, operations that zero out the memory used for storing a password after it is no longer needed can be removed by a compiler that performs dead code elimination, altering the expected program behavior [11, 73]. An analysis of the compiled program binary, on the other hand, directly applies to the code that is executed on the processor at runtime; therefore, the soundness of the analysis is not affected by optimizations. Besides modifying or removing code, the compilation process can also add new code that is not explicitly present in the source code. The usual `main` function visible to a programmer is commonly not the actual entry point of the compiled program. Instead, it is called by a statically linked library method that first calls static initializers and sets up data structures [113]. A binary analysis covers all such statically linked library code and all implicitly generated code.

Language Agnosticism. Source based analyses face several challenges of their own. High level languages usually feature a very rich syntax, and different compilers implement slightly different dialects of the same languages [16]. A common workaround for these problems is to preprocess input files into a simpler form [107]. Especially in system critical code, such as drivers or other low level components of the operating system, inline assembly code is prevalent, however. Inline assembly cannot be transformed by preprocessing and is therefore most commonly simply ignored by a source based analysis [55].

Libraries that are to be analyzed together with the main program pose a similar challenge if they are written in a different high level language. Operating on the binary avoids these issues altogether, since all source languages are translated into a hardware specific, but single target language. For languages that are compiled to bytecode, such as Java bytecode or Microsoft's Common Intermediate Language (CIL), it is already common practice to analyze bytecode instead of source, in order to avoid problems from parsing and to support all the different source languages that are available for the particular platform [67, 91].

Easy Deployment. Working with the binary also removes the need for the static analysis tool to interface with the build process of the analyzed software. Especially in large projects, it can be difficult to clearly identify all modules and source code files that are required for a complete analysis [16]. In a binary, however, all necessary components have been merged into a single executable, and the loading mechanism of the operating system can be used or emulated to retrieve all referenced dynamic libraries. This is a significant advantage for technology transfer in large companies, where it can be difficult for verification specialists in research units to obtain all source code components from product development groups.

A related issue in source code analysis is that library functions often have to be replaced by coarse grained abstractions [62]. When analyzing binaries, however, there is no fundamental difference between code of the main program and statically or dynamically linked libraries. In principle, this can even include higher level parts of the operating system. It is up to the binary analysis tool to choose the level of abstraction for the analyzed program and libraries.

Instruction Level Information. Besides the advantage of bypassing the compiler and implementation language when analyzing the binary code directly, there are scenarios where only the compiled binary can provide the necessary information: For instance, a precise execution time analysis of programs that includes the effects of caching and instruction pipelining is inevitably hardware-specific and requires knowledge about the exact instruction sequence that is being executed [56, 89, 93, 136]. Furthermore, a *dynamic* analysis monitoring the real execution of a process will observe a sequence of machine instructions, which does not easily map back to source code. In a combined analysis that merges static and dynamic results, it is therefore helpful to statically analyze the same binaries which are executed and monitored by the dynamic component [59, 61, 63, 128, 132]. Addresses of instructions then easily translate from dynamic to static analysis, and both analyses can exchange information directly. A mapping from analysis results over instructions back into source code would again face the problem of dealing with compiler optimizations, which can break the direct correspondence between blocks of instructions in the binary and syntactic elements of the source code.

Instrumentation and Whole Program Optimization. Tools that modify machine code in binaries at compile or run time can profit from static analysis as well. Anticipating future control flow in a binary can help to improve the performance and reliability of binary instrumentation toolkits [3, 92, 108], binary translators [30, 36, 127], or profilers [66, 129]. The compiler literature knows the concept of *link-time-* and *post-link-optimizers* [50, 124], which exploit the fact that the whole program including libraries and hand-written assembly routines can be globally analyzed and optimized during the final steps of the compilation process. After all libraries and modules are combined by the linker, all code is present in one file and all source languages have been translated into machine code. As in higher level compilation steps, a static analysis of the (now binary) code provides the necessary information to perform the final global optimization step for the program.

1.1.2 Analysis without Access to Source Code

The most enticing argument for performing static analysis on binaries, however, is that source code is simply not available in many practical cases and working with the binary is the only viable option.

Reverse Engineering. First of all, the information gathered by a static analysis on binaries can assist in the mostly manual process of *reverse engineering*, i.e., in recovering information about the functionality, dependencies, and interfaces of a program. A serious issue in companies with a long history of internal development of custom software is legacy software for which the original source code has been lost or which has been written directly in assembly language in the first place [54]. New requirements or changes in the environment can become a severe obstacle when the original authors of the code are no longer available. In this case, reverse engineering of the program binaries can provide the necessary information for reimplementing or patching the program.

Decompilers go even further and attempt to rebuild a close approximation of the original source code from a compiled binary [26, 37, 54, 69], commonly by making heavy use of heuristics to discover compiler idioms. While decompilation works relatively well with typed object languages, such as Microsoft CIL or Java bytecode, existing decompilers for x86 do not always provide satisfactory results, especially in presence of compiler optimizations or for programs compiled with non-standard compilers.

Another case of reverse engineering is the investigation of patent or license infringement. If source code is not available, a similarity analysis of executables can provide initial forensic data for justifying further investigation. Automated methods to detect similarities in the control flow graphs of executables [57] depend on reliable disassembly and control flow graph recovery.

Verification of Proprietary Software. The urge to protect their intellectual property often prevents software vendors to submit their products to an external analysis process. This particularly affects third-party supplied modules, such as plugins or device drivers, which are critical to the operation of a larger system. Certification programs by the framework providers, such as the Windows Logo Program [99], often rely on testing only, which cannot provide strong guarantees about the behavior of a driver or plugin. This sets the scene for another application of binary program analysis without access to source code. A static analysis on the device driver binaries can verify the conformance to API specifications, giving guarantees or uncovering bugs that can be difficult to find by testing only [8, 82]. A binary analysis can complement the usual testing of drivers without requiring active vendor support, and, depending on the surrounding legal conditions, even without vendor consent. An alternative approach to the conflict between intellectual property and verification is the use of a trusted verification protocol [25]; in contrast to binary analysis, this still requires a significant commitment by the software developer, however.

Security Analysis and Malware Detection. In sensitive environments, security audits including testing and static analysis can build confidence in the reliability of commercial software. Where source code is not available, static analysis on binaries can allow to check the software for bugs or possible hazards such as backdoors, time bombs, or other malware. On known malware, binary reverse engineering can assist in forensic analysis, and help uncover valuable information such as recipients of stolen information or control commands for botnets [2].

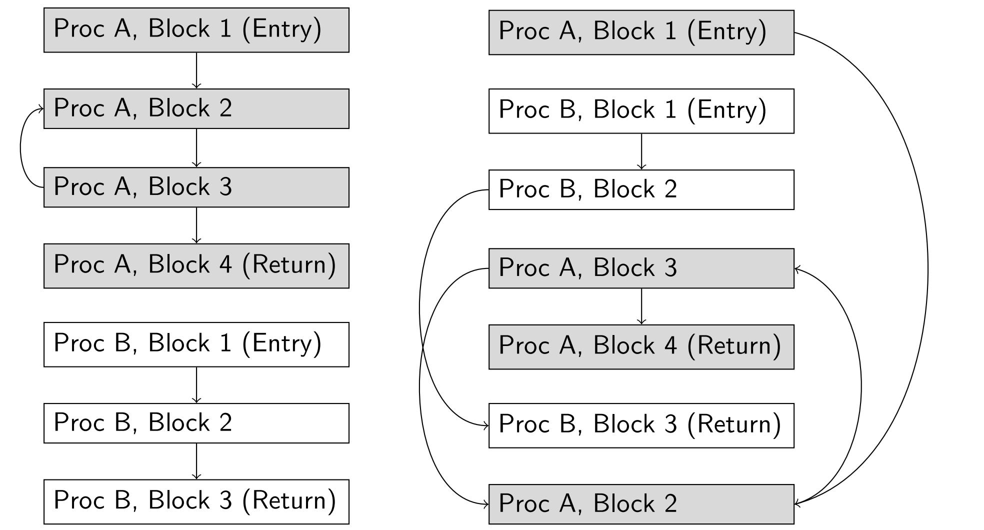
Earlier work has shown that static analysis opens the door for promising new approaches in *malware detection* [33, 79]. While classical malware detection relied on searching executables for binary strings (*signatures*) of known viruses, recent advances in the field focus on detecting patterns of malicious *behavior* by means of static analysis and model checking [33, 72, 79, 80, 84]. Such proactive approaches avoid frequent updates to signature databases and are at the same time robust against common obfuscation techniques used by poly- and metamorphic malware [32, 48].

1.2 Challenges in Binary Analysis

As briefly pointed out before, there are both scientific and engineering challenges in designing a reliable binary analysis framework. The focus of the work in this dissertation mostly lies on 32 bit x86 machine code, but many of the challenges apply to other architectures as well. Do note that some architectures make static analysis considerably easier, most notably virtual machines, such as the Java Virtual Machine (JVM), but these systems are out of the scope of this work.

Code and Data Ambiguity. There are several different ways to store binary programs on disk, such that they can easily be loaded and executed by the operating system at any time. For x86 desktop systems, the most common formats today are the Windows *Portable Executable* (PE) format [112] and the *Executable and Linking Format* (ELF), as used in Linux and other Unix variants [134]. Both formats group the file into *sections*, which can be designated to hold code, data, or both, and can be flagged to be readable, writable, and/or executable at runtime. However, the division between code and data is not strict, and code sections commonly contain data such as jump tables or string constants. In fact, nothing prevents the flagging of all sections of the compiled binary as readable, writable, and executable. Only at runtime the processor interprets some of the bytes as instructions and others as data which are processed by the instructions. The only locations inside an executable that are required to contain proper code are the entry point (i.e., main()) and, for libraries, any exported procedures. The addresses of these locations are specified in the header of the executable.

No Fixed Procedure Layout. Within the sections, the code does not have to follow a specific layout. Procedures do not necessarily follow strictly one after another (Figure 1.1(a)), but can be woven into each other, with procedure fragments connected through jumps (Figure 1.1(b)). These mangled layouts can be produced by post-processing tools that rewrite the binary as a final step after compilation and linking is complete. For instance, Microsoft's profiling and



(a) Strict layout usually produced by compilers.

(b) Mangled layout produced by post-link optimizers.

Figure 1.1: Example of possible procedure layouts in an executable.

post-link optimization framework *Basic Block Tools (BBT)* [96] uses profiling information to rearrange blocks favorably among memory pages.

Moreover, it is not even given that a procedure contains an explicit return statement: Procedures that terminate the program (e.g., by calling exit()) never return, and the tail-call optimization can replace a call followed by a return statement with a direct jump to the called procedure. Binaries that have been built directly from assembly language and have not been compiled from a high level language do not need to adhere to any concept of procedures at all.

Missing or Untrusted Symbol Information. Binaries can contain several kinds of *symbols*, which are stored in a dedicated section of the file; they are not necessary to execute the program, but provide additional information to a debugger or post-processing tools. There are two basic types of symbol information, each used for its own purpose:

- Public symbols identify exported procedures and global variables by their name and address in a binary. They are used by the (dynamic) linker to resolve calls or accesses to globals in other binaries during (dynamic) linking.

- Debug symbols for procedures and variables provide the name, type, address, and size of each static object within the procedure's address space. For blocks of machine code, they map addresses to line numbers and file names in the source code. Debug symbols are used by a debugger to present the developer with easily understandable information that directly maps back to source code.

In the release build of a typical software product, all symbols except the public symbols of dynamic libraries are *stripped*. Stripping reduces the file size of a binary and hides implementation details, providing some protection against reverse engineering. Therefore, a static analysis designed to work with publicly available binaries without access to source code must not rely on support from debug symbols. In the worst case, bad symbols might intentionally mislead an analysis. In usage scenarios where trusted symbol information is available, however, it can assist an analysis by identifying variables, procedure boundaries, and variable types.

Rich Instruction Sets. CISC (short for *Complex Instruction Set Computer*) architectures, such as x86, offer a very large number of instructions, with specialized instructions for many operations. The x86 architecture contains hundreds of instructions and thousands of possible operand combinations [74], and it continues to grow. For instance, over 300 SIMD (Single Instruction, Multiple Data) instructions have been introduced into x86 as the MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, and SSE4 extensions [115] to allow fast vector operations on multiple bytes or words at once. All of these instructions should be understood by a static analysis and at least have to be coarsely overapproximated [59]. If the implementation of an analysis simply ignores unknown instructions, it becomes inherently unsound.

Indirect Branches. One of the main problems when analyzing low level code, such as x86 assembly language, are indirect branch instructions. These correspond to goto statements where the target is calculated at runtime, or the use of function pointers combined with pointer arithmetic in high level languages. In executables, any address in the code is a potential target of an indirect branch, since in general there are no explicit labels. Failure to statically resolve the target of an indirect branch instruction thus leads to either (i) an incomplete control flow graph, where the indirect jump instruction becomes a sink, or (ii) a grossly overapproximated control flow graph, where the indirect jump is connected to every other possible instruction in the entire program. Often, data flow analysis can aid in resolving such indirect branches. Data flow analysis already requires a precise control flow graph to work on, however. This seemingly paradox situation has been referred to as an inherent "chicken and egg" problem in the literature [123, 133].

Overlapping Instructions. In Intel x86, instructions can be of variable length, unlike fixed size architectures, such as Sun SPARC, where each instruction occupies 4 bytes and is properly aligned. Each x86 machine instruction consists of an opcode, which defines the type of instruction to execute, and an optional list of operands. Operands can be registers, immediate values, or memory locations, and all take a different number of bytes to encode. The variable instruction length nature of x86 allows *overlapping instructions* (also referred to as instruction aliasing in the literature [132]): the same sequence of bytes may be interpreted by the processor as completely different instructions depending on the exact byte in which execution starts [123]. In fact, the same bytes may be executed multiple times but each time being interpreted as belonging to a different instruction. This allows to construct machine code that, as a static listing in assembly language, is mostly incomprehensible for humans. For instance, consider the

```
0000:  B8 00 03 C1 BB       mov eax, 0xBBC10300
0005:  B9 00 00 00 05       mov ecx, 0x05000000
000A:  03 C1                add eax, ecx
000C:  EB F4                jmp $-10
000E:  03 C3                add eax, ebx
0010:  C3                   ret
```

Figure 1.2: Example of overlapping instructions in x86 machine code.

```
0000:  B8 00 03 C1 BB       mov eax, 0xBBC10300
0005:  B9 00 00 00 05       mov ecx, 0x05000000
000A:  03 C1                add eax, ecx
000C:  EB F4                jmp $-10
0002:  03 C1                add eax, ecx
0004:  BB B9 00 00 00       mov ebx, 0xB9
0009:  05 03 C1 EB F4       add eax, 0xF4EBC103
000E:  03 C3                add eax, ebx
0010:  C3                   ret
```

Figure 1.3: Execution trace of the example for overlapping instructions.

fragment of machine code shown in Figure 1.2. By looking at the code, it is not apparent what the value of `eax` will be at the return instruction (or that the return instruction is ever reached, for that matter). This is due to the jump from `000C` to `0002`, an address which is not explicitly present in the listing (`jmp $-10` denotes a relative jump from the current program counter value, which is `0xC`, and `0xC` $-$ 10 = 2). This jump transfers control to the third byte of the five byte long move instruction at address `0000`. Executing the byte sequence starting at address `0002` unfolds a completely new instruction stream.

Figure 1.3 shows the instruction trace from the beginning, in the order in which it is interpreted by the CPU. After the jump, the immediate operand of the former move instruction is interpreted as the opcodes of an addition and another move instruction. The new alignment causes the former jump to become part of third addition. The new instruction sequence recombines with the original listing at address `0x00E`, and finally the execution reaches the return instruction, at which `eax` will have accumulated a value of `0xBAACC4BC`.

Abusing Calls and Returns. Another issue can arise in binaries, when instructions are used for unintended purposes: The `call` and `ret` instructions, intended for procedure calls and returns, respectively, are not required to be used for correct procedure handling. In x86, a `call` instruction simply pushes the current program counter onto the stack and jumps to the given target. Conversely, the `ret` instruction pops an address from the stack and jumps to it. However, a `ret` instruction can just as well be used for an indirect jump: The instruction `jmp eax` executes the same jump as the sequence `push eax; ret`. As a consequence, call and return instructions cannot generally be treated equivalently to procedure invocations and returns in high level languages. The concept of *return oriented programming* [22, 125] uses this behavior of return instructions in vulnerability exploits to create chains of program code that together constitute a malicious code sequence. This technique is particularly effective if used together with overlapping instructions [125].

Of course, code that misuses calls and returns or exploits overlapping instructions will never be generated by usual compilers. Intentionally obfuscated, handcrafted assembly code is prevalent in sophisticated malware or other software protected against reverse engineering, however, and can use these techniques to thwart automated and/or manual analysis. A robust analysis method applicable to machine code thus has to be able to correctly handle these cases.

Lack of Types. With debug symbols generally not available in binaries, an analysis has no type information at its disposal. Global and local variables, arrays, and records all uniformly appear as addresses indexing the large continuous array that is the virtual memory available to a process. The type that a variable or structure had in source code is no longer visible after compilation, and the compiler commonly reuses the same register or stack location for variables of different types, which makes a classical flow insensitive type inference analysis impossible. Different types can occupy a different amount of bytes, therefore a new value assigned to a register or memory location can overlap with an earlier value, overwriting some but not all of the bytes.

Pointer Aliasing. A consequence of the lack of types and the a priori unknown control flow is that a cheap points-to analysis is made impossible. Every dereference of an unknown pointer can mean an access to any memory address, be it the stack, global memory, or the heap. A write access then causes a *weak update* to the entire memory: After the write, every memory location *may* contain the written value, which dramatically impacts the precision of the analysis. In particular, weak updates potentially overwrite return addresses stored on the stack (or function pointers anywhere in memory), which can cause spurious control flow to locations that are never executed at runtime. The goal of a sound and precise analysis on binaries is thus to achieve *strong updates* wherever possible: If a pointer can only point to one specific address in a state, the targeted memory location *must* contain the written value after a write access [28].

Self Modifying Code. Finally, a notorious challenge in analyzing binaries is *self-modifying code*: Machine code is generated at runtime, possibly overwriting earlier code at the same address, and executed afterwards. In that case, the instructions eventually executed are not even present in the file. As the aforementioned techniques, self-modifying code is particularly popular with malware and obfuscated software; however, the same behavior is also present in just-in-time compilers or emulators, which translate and execute machine code on the fly.

1.3 Traditional Disassembly and Analysis

The classic setup for binary analysis, which is commonly encountered in the literature [7, 31, 34, 95], is to use a stand-alone *disassembler* to preprocess the binary and make it easily parseable by the static analyzer. A disassembler is a low level debugging and reverse engineering tool that generates a plain text listing of the assembly code equivalent to the machine code in the binary. Note that disassemblers are very different from decompilers: for the most part, a disassembler directly translates code bytes into assembly mnemonics, i.e., textual representations of the machine instructions, whereas a decompiler attempts to generate high level language source code (such as C or Java) from a binary. An exact definition of the disassembly problem is somewhat elusive, as the main job of disassemblers is to aid human engineers in understanding executable code. A minimal and purely syntactic definition can be given as follows:

Definition 1.1 (Disassembly) *The disassembly problem is to generate from an executable a listing in assembly language such that a given assembler will encode the listing to an executable syntactically equivalent to the original one.*

This definition is parameterized by an *assembler*, which defines the syntax of its supported *assembly language* and a method of translating assembly language programs into executables. Note that by this definition, the result of disassembly is by no means unique. In particular, a trivial solution would be a listing defining all bytes in the binary file as constants using, say, the `db` construct in assembly language. Usually, a disassembler will make a best-effort approach to decode as many bytes into instructions as possible. Note further that the instructions visible in the output listing are not necessarily ever executed. Overlapping instructions, as discussed above, or data misinterpreted as code can produce pseudo-instructions that will never execute at runtime.

All disassemblers translate binary machine code into instruction mnemonics using lookup tables. CISC architectures and variable instruction length make this a tedious, but still straightforward task. Traditionally, the challenge and main design choice in implementing a disassembler lies in how to trace the control flow to decode sequences and branches of code. Linn and Debray [90] identified two basic strategies for disassemblers:

- *Linear sweep* sequentially decodes bytes into instructions from the beginning of the first section of an executable until the end of the file. This simple strategy, used in tools such as GNU objdump, is able to produce correct disassembly according to Definition 1.1, but it is of very limited practical use for disassembling entire executables. Linear sweep easily loses the correct alignment of instructions because of data or padding bytes between code blocks. Due to overlapping instructions, misalignment can lead to an alternate sequence of instructions that does not reflect the instructions that are actually executed at runtime. Alternate instruction streams that are a consequence of misalignment have a tendency to realign with the correct stream after few instructions [120]; together with the fact that the x86 instruction set is so densely coded that most byte sequences constitute valid code, this can make disassembly errors introduced by misalignment hard to spot.

- *Recursive traversal* disassemblers start at the entry point of the file, interpret branch instructions, and decode the program by depth first search, translating bytes actually reached by control flow. This allows the disassembler to skip over data bytes mixed into code sections. On the downside, this strategy is not guaranteed to process all bytes in the executable, since not all code locations are accessed through direct branches from the entry point. Function pointers, callbacks, and other indirect branches can obscure the control flow in the executable, hiding code from simple syntactic recursive traversal.

  To avoid this problem, state-of-the-art disassemblers usually augment recursive traversal by heuristics to detect potential pieces of code in the executable. These heuristics exploit the presence of known compiler idioms, such as recurring procedure prologues or common patterns in the calculation of switch-jumps from jump tables [66].

Today's de facto industry standard for disassembly is IDA Pro, which follows the recursive traversal strategy. Its heuristic looks for common prologue bytes generated by compilers to identify procedure entry points. For instance, a common x86 sequence to set up the frame pointer for the current procedure's stack frame is `push ebp; mov ebp, esp`. Procedures not starting with a standard procedure prologue can thus be missed if they are invoked only through function pointers or indirect jumps, i.e., if their address is not an explicit operand of a control flow instruction. A standard assumption made by recursive traversal disassemblers, including IDA Pro, is that every call eventually returns to its fall-through successor. For calls to procedures that never return because of a call to exit or a similar method, this assumption can cause IDA Pro to decode instructions directly following the call that are never executed or belong to a different procedure.

In a toolchain that uses an external disassembler, the disassembler takes care of decoding bytes into instruction mnemonics and operands. The static analyzer is constructed as a separate tool and processes the listing produced by the disassembler. From the viewpoint of static source code analysis, this separation of concerns appears natural at first; parsing the assembly listing then simply replaces parsing of a high level language source code file. Yet, available commercial disassemblers, such as IDA Pro, are built for aiding humans in the debugging or reverse engineering process. In particular, IDA Pro – short for *Interactive Disassembler* – is meant to be used interactively, with the human engineer resolving misinterpretations of data as code or providing additional entry points. For an automated analysis of binaries, such an interactive approach to disassembly is not an option.

A static analysis typically uses the control flow graph (CFG) [1] of a program to compute abstract states. In source based analyses and executable analyses that use external disassemblers, the CFG is built by parsing the code listing and looking up the targets of branches and procedure calls. Unfortunately, the CFGs built from an assembly file generated by a heuristics-driven recursive traversal disassembler can have many disconnected components. If indirect jumps have unknown targets, or if there are callback methods passed to the system that are only invoked by external library methods, some code blocks in the disassembled binary will appear not to be referenced from anywhere. Similarly, the indirect jump or call instructions in the CFG will have no successors. Thus, any static analysis that uses such graphs as an initial overapproximation is unsound, as edges are missing from the CFG.

Apparently, a toolchain for static analysis on executables does not require a solution to the disassembly problem, but rather a high fidelity control flow graph for the executable. We therefore now define the concept of *control flow reconstruction*, which is more restrictive than the earlier definition of disassembly and tailored directly to the requirements of static analysis.

Definition 1.2 (Control Flow Reconstruction) *Control flow reconstruction is the problem of determining an overapproximation of all possible sequences of program locations (addresses) that will be executed by a given program.*

Typically, a solution to this problem will be given as a graph or automaton encoding possible control flow in the program.

1.4 Overview on the Proposed Method

This dissertation introduces a novel and theoretically well founded approach to disassembly, control flow reconstruction, and static analysis of x86 binary executables. At the core lies an integrated disassembly and analysis loop, which defines an abstract interpretation of the binary executable. Figure 1.4 lists all the challenges identified in Section 1.2 and relates them to the components of the approach. Each of the components is covered in a separate chapter of this dissertation.

Intermediate Language. The problem of dealing with large instruction sets is addressed by translating instructions into a low level intermediate language (IL), which is introduced in Chapter 2. The IL is designed specifically for "upward" translation from machine code, in contrast to intermediate representations used by compilers that are designed for "downward" translations from high level languages. The IL breaks complex assembly instructions into a sequence of statements that capture the semantics of the machine code. Specifications for this translation can be written using the semantics specification language (SSL) introduced by Cifuentes and Sendall [38]. In the course of the translation, call and

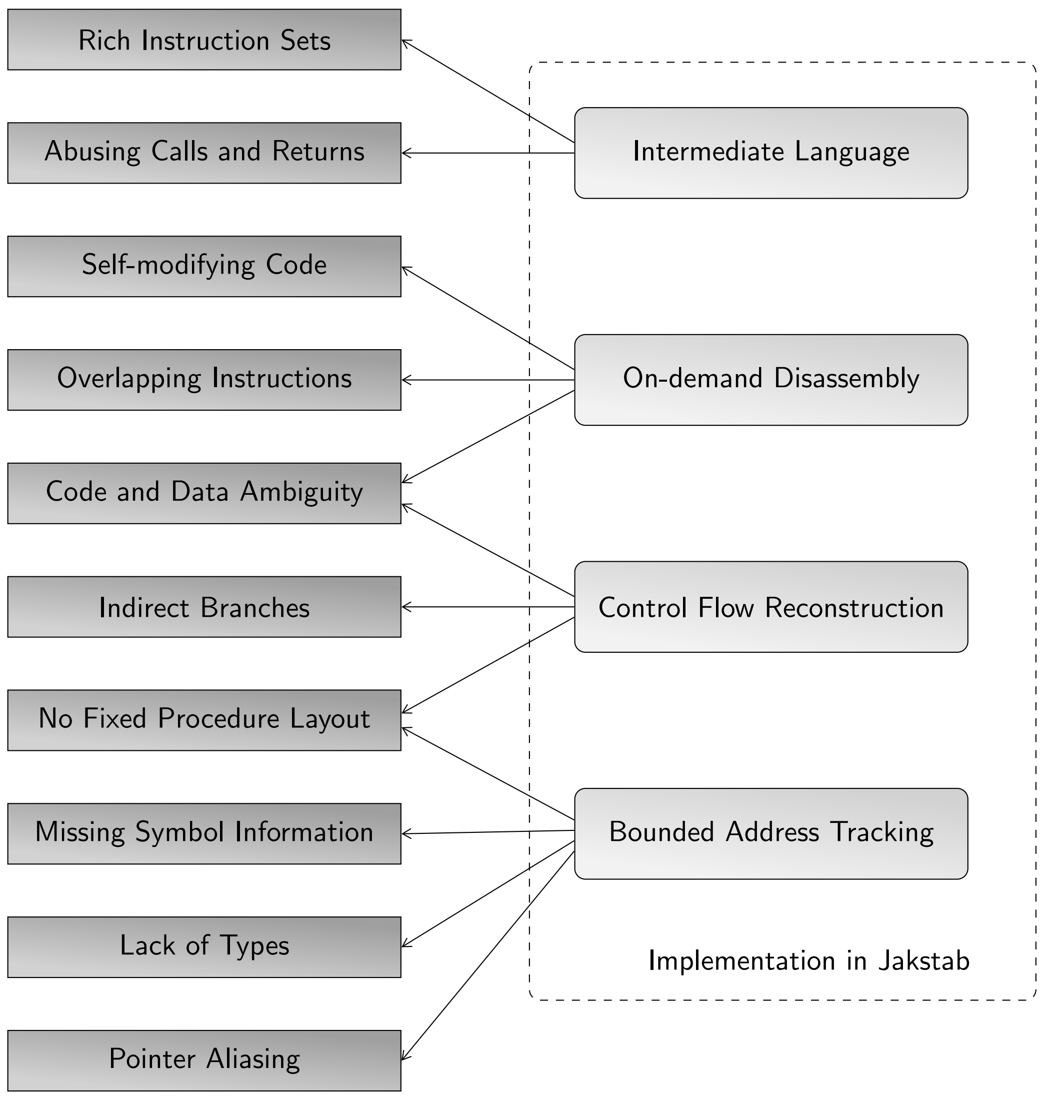


Figure 1.4: Challenges in binary analysis and the proposed solutions.

return instructions are translated into stack accesses and (possibly indirect) goto statements. This nullifies any obfuscating effects of abusing call and return instructions and allows to treat both instruction types equally.

Control Flow Reconstruction. The non-obvious control flow and structure of binaries and the seeming "chicken and egg" problem of using data flow analysis to resolve indirect branches are a major challenge for static analysis. Chapter 3 introduces a formal framework based on abstract interpretation that integrates control and data flow analysis on low level programs (i.e., executables represented by IL statements). The framework uses a special operator to resolve the targets of jump statements and transforms them into labeled control flow edges. It is not fixed in the type of abstract domain for data flow analysis; if the domain satisfies certain conditions, the approach is guaranteed to determine the most precise overapproximation of the program's control flow with respect to the abstract domain. It is further proven that this result holds independently of the precise order in which control and data flow information is calculated. In contrast to earlier structural [78] or heuristic [85, 105, 120] approaches, this rigorous framework for control flow reconstruction is not affected by a fragmented layout of procedures and distinguishes code from data bytes by determining an overapproximation of the set of possible program counter values.

Bounded Address Tracking. The lack of reliable symbol information and types for variables and structures in executables aggravates the aliasing problem for static program analysis. Supplementing the generic control flow reconstruction framework, Chapter 4 introduces Bounded Address Tracking, a highly precise abstract domain that models registers and memory locations as both pointers and integer values and maintains path sensitivity. The underlying memory model, which is inspired by VSA [7], partitions the memory into separate regions. Every value is tagged with a region identifier, which serves as a symbolic base address. Pointers to the global memory region, the stack, and the heap can thus be identified and are assumed to not overlap. Integers are tagged with the global memory region, as it corresponds to a zero base address. Path sensitivity allows the analysis to perform context sensitive analysis of procedure calls, without assuming a correct layout or behavior of procedures. It is a prerequisite for treating return values just like any other value stored on the stack. With this approach, however, even modifications of the return address are precisely modeled.

Termination of the analysis is assured by imposing a bound over the number of values tracked per variable per location. If a variable exceeds the bound, its values are widened in two steps. Handling pointers and integers within the same domain accounts for the low level nature of assembly code and provides a solution to the lack of types. Pointer aliasing is minimized by the high precision of the domain, which avoids overapproximation of pointers and hence weak updates, as long as the number of targets remains below the definable bound.

On-Demand Disassembly. Closely tied to the proposed approach to control flow reconstruction is the idea of on-demand disassembly, which is an essential part of the software architecture for binary analysis discussed in Chapter 5. Instead of attempting to disassemble as many instructions as possible in a separate preprocessing step, only a single instruction is disassembled at a time. In essence, the instruction fetch is considered part of the abstract interpretation, and thus only the instruction relevant for the next execution step is decoded. This allows to deal with overlapping instructions, as no fixed representation is required that maps every byte uniquely to a single instruction. Instead, the same bytes can be interpreted as different instructions depending on execution context. In a similar manner, self-modifying code can be dealt with, by disassembling bytes from the current abstract memory state (although this is not currently implemented).

Implementation in Jakstab. Chapter 5 further presents the disassembler and static analysis tool Jakstab, which implements the concepts introduced in this dissertation. Jakstab is written in about 40 KLOC of Java and is able to process both Windows and Linux executables, with its primary focus lying on the Windows family of operating systems. Architecture-wise, the current implementation supports only x86 code (although it is designed to be extendable to other architectures); the underlying methods are not platform specific, however. It reconstructs the control flow of a binary by exploring the reachable state space, and is able to check specifications in the form of invariant assertions introduced in an environment model for the program.

The implementation in Jakstab follows the concept of *Configurable Program Analysis* (CPA) by Beyer et al. [17], which defines a practical interface to configure and combine reachability analyses. The original CPA algorithm has been modified to follow the framework of Chapter 3 by integrating the resolve operator in the form of a call to a transformer factory that provides control flow edges. Depending on the intended application of the analysis, the strict soundness provisions of the framework can be deliberately weakened by using different available transformer factories that make assumptions about the program behavior.

Besides Bounded Address Tracking, a number of additional, classical abstract domains have been implemented in Jakstab. Constant propagation, call stack analysis, forward expression substitution, and live variable analysis, are classical textbook analyses that have been defined as CPAs and adapted to low level IL programs. A composite analysis uses a default strategy of merging information from different analyses. Code transformations allow to simplify the program after control flow reconstruction, and a second round of analyses can be run on the reduced program.

Experiments. Chapter 6 presents experimental results for two different application scenarios. The first part contains a study of verifying API usage specifications on device driver binaries. The results from analyzing several drivers from the Windows driver development kit are compared against the state-of-the-art approach by Balakrishnan and Reps [8]. They show that Jakstab using Bounded Address Tracking yields less false positives and is considerably faster, all without making unsound assumptions as part of the disassembly process. Applicability to real world binaries without access to source code or symbols is demonstrated by running Jakstab on all drivers (over 300) installed on a regular desktop PC.

A second study demonstrates another application scenario, where Jakstab is configured to use heuristics and assumptions to cover as many instructions as possible in a manner similar to the commercial disassembler IDA Pro. For analysis, only the simple and fast constant propagation is activated, which aids in resolving call addresses cached in registers. The results show that its capabilities as a disassembler are comparable to IDA Pro, and that constant propagation can suffice to exceed IDA Pro's ability in resolving call targets.

1.5 Contributions

Summarizing the above outline, this dissertation makes the following contributions to the state of the art:

- The design of an abstract interpretation-based, integrated control and data flow analysis framework for low level binary code, giving a solution to the open problem of optimal control flow reconstruction from binary executables (Chapter 3).
- The introduction of Bounded Address Tracking, a very precise abstract domain for combined pointer and value analysis (including pointer arithmetic), that generally allows strong updates to be performed up to a tunable bound (Chapter 4).
- Embedding on-demand disassembly, control flow reconstruction, and multiple analyses into an extensible program analysis framework working on binaries (Chapter 5). The framework is configurable in several aspects to allow a wide range of analyses, from sound abstract interpretation to heuristics-supported disassembly.

- Showing the feasibility of the approach and improvements over existing approaches by conducting a case study on the analysis of Windows device driver binaries. A second study compares disassembly and control flow reconstruction results with the commercial disassembler IDA Pro (Chapter 6).

Parts of this dissertation have been published as [81] (disassembly augmented by constant propagation), [83] (joint control and data flow analysis), and [82] (Bounded Address Tracking and experiments on driver binaries).

Chapter 2

An Intermediate Language for Executable Analysis

This chapter introduces the intermediate language (IL) and related concepts that will be used for analyzing executables throughout the remainder of the dissertation. ILs are a concept common in compiler design, where they help abstracting from a particular source language and act as a connecting layer between the high level source language and the target machine code. The compiler performs most of its static code analysis on the IL, allowing the analysis to be mostly architecture- and language-independent. Similarly, ILs can allow a binary analysis to abstract from machine code and to formulate the analysis in an architecture-independent manner.

2.1 Overview

CISC architectures such as x86 offer very rich instruction sets. In these architectures, a single instruction can affect multiple registers and status flags and can even represent non-trivial operation sequences including loops (e.g., using the `repnz` prefix). The somewhat naive, direct approach is to deal with the hundreds of different instructions directly, by hand-coding abstract transformers/ transfer functions for some or all instructions, as implemented in the original

CodeSurfer/X86 tool [7], for example. This process is extremely tedious and error prone, as it requires reimplementing the hundreds of transformers for each new analysis if all instructions are to be correctly supported. A significantly less cumbersome approach, which will be used in this work, is to define translations from assembly instructions to an IL and to specify transformers only in terms of the intermediate language. From a specification of the instruction semantics, assembly instructions are translated into sequences of low level IL statements. For example, the instruction `push eax`, which pushes the contents of register `eax` to the stack and decrements the stack pointer, translates to the IL code sequence $m_{32}[esp] := eax; esp := esp - 4$. The syntax of the IL was inspired by the semantics specification language (SSL) by Cifuentes and Sendall [38].

The low level nature of machine code influenced the design of the language and the choice of IL statements available. For instance, x86 machine code does not contain explicit, structured conditional statements, but instead uses conditional jumps. Conditional execution of code blocks is realized by first comparing two operands using a comparison instruction such as `cmp` or `test`, which sets the flags according to the result of the comparison. The flags then decide whether a later conditional jump instruction is taken or not.

Consider the example below. The high level code on the left assigns the minimum of two variables x and y to the memory location pointed to by p. It is translated by compilers to assembly code such as the one shown on the right, if x, y, and p are allocated to registers `eax`, `ebx`, and `edx`, respectively.

```
if (x > y) {                     cmp eax, ebx
    x = y;                       jle label
}                                mov eax, ebx
*p = x;                   label: mov dword ptr [edx], eax
```

The first instruction, `cmp eax, ebx`, subtracts `ebx` from `eax`, sets the status flags according to the result of the subtraction, and discards it. The instruction translates to the IL code

$$\begin{aligned}
CF :=& (eax <_u ebx) \\
OF :=& (eax < 0 \land ebx \geq 0 \land eax - ebx > 0) \lor \\
& (eax \geq 0 \land ebx < 0 \land eax - ebx < 0) \\
SF :=& (eax - ebx < 0) \\
ZF :=& (eax = ebx).
\end{aligned}$$

Here, $<_u$ denotes unsigned comparison, $-$ denotes bit-vector subtraction, *CF* is the carry flag, *OF* the overflow flag, *SF* the sign flag, and *ZF* the zero flag. The second instruction of the conditional idiom is the conditional jump `jle label`, meaning "jump if less or equal". It translates to the *guarded jump*

$$\texttt{if } ((SF \veebar OF) \lor ZF) \texttt{ jmp label},$$

which evaluates the flags and transfers control to `label`, if the condition is met ($\veebar$ denotes exclusive or). The sign flag signals that the result was negative, i.e., the first operand was less than the second operand; the overflow flag signals that the result was negative but smaller than the smallest representable number and overflowed into the positive range. Both flags together conversely signal that the result was too large and overflowed into the negative. The zero flag finally covers the case that both numbers are equal.

If the condition for the jump is not met, the body of the if-clause is executed, which translates to the assignment $eax := ebx$. From the body, execution falls through to the next statement after the if-clause, which is also the target of the conditional jump. The pointer dereference and assignment to a double word (`dword` in Intel assembly syntax) translates to $m_{32}[edx] := eax$ in the IL syntax.

2.2 Syntax

To reduce the complexity of implementing abstract transformers for the IL, the syntax is kept as simple as possible, with only a small set of allowed expressions and statements that capture the low level aspects of assembly language. The statements are grouped into two families. The basic statements represent direct effects of individual instructions such as register assignments. The abstract statements represent higher level concepts such as memory allocation.

2.2.1 Expressions

The set **Exp** of expressions of the IL contains common arithmetic, Boolean, and bit-manipulation operations. All arithmetic operations are operations on bit-vectors, i.e., depending on the bit-vector length, they can cause over- or underflow (see also the definition of IL types in Section 2.3). **Exp** is given by the following BNF grammar:

$$
\begin{aligned}
<\!\mathit{expr}\!> &::= <\!\mathit{num}\!> \mid <\!\mathit{var}\!> \mid \mathrm{pc} \mid <\!\mathit{memloc}\!> \mid <\!\mathit{nondet}\!> \mid <\!\mathit{unary\_op}\!>\ <\!\mathit{expr}\!> \mid \\
&\qquad <\!\mathit{expr}\!>\ <\!\mathit{binary\_op}\!>\ <\!\mathit{expr}\!> \mid <\!\mathit{cond}\!> \mid <\!\mathit{bit\_extr}\!> \mid <\!\mathit{extend}\!> \\
<\!\mathit{num}\!> &::= (-)^{?}(0-9)^{+} \\
<\!\mathit{var}\!> &::= (a-z \mid A-Z)^{+} \\
<\!\mathit{memloc}\!> &::= \mathrm{m}_{<\mathit{num}>}[<\!\mathit{expr}\!>] \\
<\!\mathit{nondet}\!> &::= \mathrm{nondet}(<\!\mathit{num}\!>) \\
<\!\mathit{unary\_op}\!> &::= \neg \mid - \mid \\
<\!\mathit{binary\_op}\!> &::= \ < \ \mid \ \leq \ \mid \ <_u \ \mid \ \leq_u \ \mid \ = \ \mid \wedge \mid \vee \mid \veebar \mid + \mid \cdot \mid \div \mid \mathrm{mod} \mid \ggg \mid \gg \mid \ll \mid \\
&\qquad \mathrm{rol} \mid \mathrm{ror} \\
<\!\mathit{cond}\!> &::= <\!\mathit{expr}\!>\ ?\ <\!\mathit{expr}\!> : <\!\mathit{expr}\!> \\
<\!\mathit{extend}\!> &::= (\mathrm{sgnex} \mid \mathrm{zeroex})\ <\!\mathit{expr}\!>\ <\!\mathit{expr}\!> \\
<\!\mathit{bit\_extr}\!> &::= <\!\mathit{expr}\!>@[<\!\mathit{expr}\!> : <\!\mathit{expr}\!>]
\end{aligned}
$$

Apart from the usual arithmetic operators, the grammar of expressions accepts the following non-standard operations:

- *nondet*: To model input from the hardware, expressions can contain the keyword *nondet*, which nondeterministically evaluates to some bit-vector value of the supplied bit length in its concrete semantics.
- sgnex: Casts an expression (second parameter) to a greater bit length (first parameter) maintaining the sign, i.e., adds leading zeros to positive numbers and leading ones to negative numbers.
- zeroex: Casts an expression (second parameter) to a greater bit length (first parameter) by filling the new bits with zeros, ignoring the sign.
- $e@[a : b]$: Casts an expression e to a smaller bit length by extracting only the bits a through b.
- $<_u$, $\leq_u$: Unsigned comparison.
- $\ggg \mid \gg \mid \ll$: Bitwise arithmetic right shift maintaining the sign, right shift, and left shift, respectively.
- ror, rol: Bitwise right and left rotation.

An IL program uses a finite set V of processor registers and temporary variables, which can be necessary for specifying instruction semantics. For simplicity, registers and temporary variables will be referred to simply as *registers* in the future. The program counter *pc* is a separate syntactic element but can be used in expressions just like regular registers. The store (memory) is accessed through expressions $m_b[expr]$, where the integer b denotes the number of bits accessed from the address given by expression *expr*. The term *variables* will be used to refer to both registers and memory locations when a distinction is not necessary.

2.2.2 Basic Statements

Assembly instructions directly translate to sequences of statements from the following set of four basic IL statements:

- Register assignments $v := e$, with $v \in V$ and $e \in \mathbf{Exp}$, assign the value of expression e to register v. This includes assignments to flags, which are treated as separate single-bit registers in the IL (in x86 processors, the flags are individual bits of the EFLAGS register).
- Store assignments $m_b[e_1] := e_2$, with $e_1, e_2 \in \mathbf{Exp}$, assign the value of expression e_2 to the b bit memory location at the address computed by evaluating e_1.
- Guarded jumps of the form $\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2$, with $e_1, e_2 \in \mathbf{Exp}$, transfer control to the target address resulting from evaluating e_2, if the guard expression e_1 does not evaluate to 0_1. Otherwise, they do nothing.
- halt statements terminate execution.

Note that call and return instructions receive no special treatment but are translated to assignments and jumps in the IL. In x86 assembly these instructions simply store the current program counter on the stack and jump to a target, or read a return address from the stack and jump to it, respectively. There is no fixed concept of procedures in x86 assembly, so relying on binary code to respect high level structuring into procedures can introduce unsoundness into the analysis. This makes the IL especially well suited to represent code protected against disassembly, including malicious code. For example, malicious code commonly misuses return instructions as generic jumps by pushing the desired target address on the stack immediately before executing a return. Anti-disassembly patterns like this thwart traditional recursive traversal disassemblers [90] that assume code to be produced by well-behaved compilers.

2.2.3 Abstract Statements

Besides the basic statements, the IL also offers a set of abstract statements that do not correspond to regular assembly instructions. Instead they can be used to abstract certain behavior of the execution environment.

- Allocation statements alloc v, e, with $v \in V$ and $e \in \mathbf{Exp}$, reserve a new block of memory on the heap and store its address in a register.
- Deallocation statements free v with $v \in V$ release the block of memory pointed to by the provided register.
- Statements assume e with $e \in \mathbf{Exp}$ ensure a condition in all executions following the statement.
- Assertions assert e with $e \in \mathbf{Exp}$ indicate an error if a condition is not met in executions passing through the statement.
- Statements havoc $v <_u n$, with $v \in V$ and a bound n, non-deterministically assign a value smaller or equal to n to a register. It is similar to an assignment of the *nondet* expression, but can be used for guiding the abstraction of nondeterminism (see Section 4.5).

None of these statements can be generated directly from x86 assembly code. They are, however, useful for verifying specifications and for abstracting functions of runtime libraries or the operating system. The assume statement will be generated during control flow reconstruction (see Chapter 3). The other abstract statements are obtained from binaries defining the abstract environment by using specially crafted illegal instruction sequences (details of the encoding are discussed in Section 6.1.3).

2.3 Types

Even though most machine code, including x86, does not have a type system even remotely resembling those of high level languages, the operands of instructions can be of different size, e.g., bytes, words (two bytes), and double words (four bytes). These low level types are reflected in the IL. All expressions in the IL are of a bit-vector type of some finite length. Expressions are defined only for operands of equal bit lengths, with the exception of zero extension, sign extension, and bit extraction.

The set of all bit-vectors of finite length is denoted by $\mathbb{I}_*$, the set of all bit-vectors of some particular length b is denoted by $\mathbb{I}_b$. The notation for bit-vector constants used in the following is n_b, where n is the integer value of the constant, and b is the length (number of bits) of the bit-vector, e.g., 91_8 for the bit-vector 01011011. Where ever the bit length b of a register v is relevant, it will be denoted by the similar syntax v_b. Expressions of bit length 1 take the place of Boolean expressions, Boolean **true** and **false** are represented by the single-bit bit-vectors 1_1 and 0_1, respectively. Negative integers are represented as the two's-complement of their absolute value (so, 1_1 represents a value of -1 in a single bit). The sign is thus determined by the most significant bit of each bit-vector value.

Note that floating point values and expressions are omitted from the IL, since the analyses presented later in this dissertation (Section 5.3) do not have specific support for floating point arithmetic. In principle, it is of course possible to extend the IL to include floating point types and expressions. In fact, the specifications used by Jakstab do include floating point expressions, which are safely overapproximated to unknown values in the analyses considered.

2.4 Semantics

The concrete semantics of the IL is defined in terms of states $S = \mathbf{Loc} \times \mathbf{Val} \times \mathbf{Store} \times \mathbf{Heap}$, consisting of the location valuation $\mathbf{Loc} := \{pc\} \to \mathbf{L}$, the register valuation $\mathbf{Val} := V \to \mathbb{I}_*$, the store valuation $\mathbf{Store} := \mathbb{I}_* \to \mathbb{I}_*$, and a heap set $\mathbf{Heap} := \mathbb{I}_* \to \mathbb{I}_*$, which maps addresses of allocated heap objects to their corresponding sizes. Heap blocks are allocated between some constant address h_0 and a maximum address h_{max} in the store. Allocation of heap objects in a single, continuous store constitutes the *flat memory model*. Accesses to parts of a state $s \in S$ are denoted by

- $s(pc)$: the value of the program counter in s,

- $s(v)$: value of register v,
- $s(m_b[x])$: value of the b bit memory location at address x, and
- $s(H)$: the current heap set.

The syntax $s[x \mapsto y]$ denotes the state obtained by updating a part x (i.e., the pc, a register, a memory location, or the heap set) of state s with a new value y.

The concrete semantics is then given by the concrete **post** operator mapping states and statements to states in Table 2.1. It uses the operator $\textbf{eval} : \textbf{Exp} \rightarrow \mathbb{I}_*$ to concretely evaluate IL expressions. The concrete semantics of expression evaluation is not formally defined here, but is assumed to follow the usual interpretation in high level programming languages. Non-standard expressions have been described in Section 2.2.1. Note that the concrete semantics introduced here corresponds to a forward collecting semantics [44, 46, 47], which describes the reachability of program states. This concrete semantics precisely characterizes the feasible control flow of a program, which is the program property of interest for control flow reconstruction. Throughout this dissertation, this concrete semantics will therefore serve as the reference point that describes the most precise static control flow information.

The concrete semantics for the basic statements is straightforward. Assignments to registers update the state with the right-hand side of the assignment evaluated in the current state. Similarly, memory assignments update the store at the address resulting from evaluating the address expression of the memory access. Guarded jumps do nothing but fall through to the next statement if the guard expression evaluates to 0_1, otherwise they transfer control to the location determined by evaluating the target expression. Halt statements terminate execution and have no successor state, denoted by $\perp$.

The concrete semantics for abstract statements is designed to capture specific aspects of the execution environment of an executable. The allocation statement simulates the behavior of a simple malloc implementation. The heap is defined to start above some limit address h_0 and extend until h_{max}, and an invocation

AssignReg

$$\mathbf{post}[\![[v := e]^{\ell}_{\ell'}]\!](s) := s[v \mapsto \mathbf{eval}[\![e]\!](s)][pc \mapsto \ell']$$

AssignMem

$$\mathbf{post}[\![[m_b[e_1] := e_2]^{\ell}_{\ell'}]\!](s) := s[m_b[\mathbf{eval}[\![e_1]\!](s)] \mapsto \mathbf{eval}[\![e_2]\!](s)][pc \mapsto \ell']$$

GuardedJump

$$\mathbf{post}[\![[\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]^{\ell}_{\ell'}]\!](s) := \begin{cases} s[pc \mapsto \ell'] & \text{if } \mathbf{eval}[\![e_1]\!](s) = 0 \\ s[pc \mapsto \mathbf{eval}[\![e_2]\!](s)] & \text{otherwise} \end{cases}$$

Halt

$$\mathbf{post}[\![[\mathsf{halt}]^{\ell}_{\ell'}]\!](s) := \bot$$

Alloc

$$\mathbf{post}[\![[\mathsf{alloc}\ v, e]^{\ell}_{\ell'}]\!](s) := \mathsf{let} z = \mathbf{eval}[\![e]\!](s), h > h_0 \text{ minimal such that}$$
$$\forall (h', z') \in s(H). h \geq h' + z' \vee h + z \leq h'$$
$$\begin{cases} s[v \mapsto h][H \mapsto H \cup (v, z)][pc \mapsto \ell'] & \text{if } h + z \leq h_{max} \\ \bot(\text{raise error}) & \text{otherwise} \end{cases}$$

Free

$$\mathbf{post}[\![[\mathsf{free}\ v]^{\ell}_{\ell'}]\!](s) := s[H \mapsto H \setminus (v, \cdot)][pc \mapsto \ell']$$

Assume

$$\mathbf{post}[\![[\mathsf{assume}\ e]^{\ell}_{\ell'}]\!](s) := \begin{cases} \bot & \text{if } \mathbf{eval}[\![e_1]\!](s) = 0 \\ s[pc \mapsto \ell'] & \text{otherwise} \end{cases}$$

Assert

$$\mathbf{post}[\![[\mathsf{assert}\ e]^{\ell}_{\ell'}]\!](s) := \begin{cases} \bot(\text{raise error}) & \text{if } \mathbf{eval}[\![e_1]\!](s) = 0 \\ s[pc \mapsto \ell'] & \text{otherwise} \end{cases}$$

Havoc

$$\mathbf{post}[\![[\mathsf{havoc}\ v <_u n]^{\ell}_{\ell'}]\!](s) := s[v \mapsto x][pc \mapsto \ell'], \text{ with some } x \leq n$$

Table 2.1: Concrete semantics of the intermediate language.

of alloc v, e assigns to the designated register v the lowest address h such that there is sufficient free memory above h to fit the requested size z calculated from evaluating expression e. The heap set H, which is defined to be part of the concrete state, then stores the fact that a block of memory of size z is allocated at address h. If not sufficient free memory is available, an error is signaled. The abstract statement free removes allocated addresses from H again, freeing the corresponding memory block.

Assume statements are used to "filter" executions; they do nothing, if the condition being assumed evaluates to 1_1 in the current state, but if it evaluates to 0_1, they terminate the current execution (again denoted by the successor state $\perp$). Assertion statements behave similarly, except that they signal an error if the execution is terminated. Assertion statements are used when not meeting the asserted condition indicates an error, i.e., assertions express specifications for the program.

Havoc statements havoc $v <_u n$ nondeterministically assign a value x with $0 \leq_u x \leq_u n$ to v. In the concrete semantics, the same effect can be achieved by the sequence of expressions $v := \mathit{nondet}$; assume $v \leq_u n$. In an abstract semantics, however, the two different methods for introducing nondeterminism into a program can be abstracted differently, effectively allowing to select different levels of abstraction by choosing the appropriate statement. This will be further discussed in Section 4.5, which motivates this essentially annotation based abstraction for a specific analysis.

2.5 IL Programs

By translating every machine code instruction from an executable into statements of the IL before feeding it to the analysis, it is possible and practical to view binary executables as IL programs.

The set of basic and abstract statements is denoted by by **Stmt**. The set of program *locations* $\mathbf{L} \subseteq \mathbb{I}_*$ is the set of bit-vectors of some length specified by the architecture (in the case of 32 bit x86 code, it is 32). An IL program is then given as the tuple $\langle V, \mathbf{CodeData}, \mathbf{start}\rangle$, with the set of registers V, the unique starting location **start**, and $\mathbf{CodeData} := \mathbf{L} \rightarrow (\mathbf{Stmt} \times \mathbf{L} \times \mathbb{I}_*)$. The finite mapping **CodeData** expresses that program locations can be interpreted as either code or data. To each program location, it maps

1. the statement obtained from interpreting the location as code,
2. the address of the logically next statement, if the location is interpreted as code, and
3. the bit-vector value obtained from interpreting the location as data.

The code interpretation of locations in **CodeData** is denoted by $[stmt]^{\ell}_{\ell'}$ for $\ell \mapsto (stmt, \ell', \cdot) \in \mathbf{CodeData}$, which follows the notation from standard program analysis literature [109]. Where the location of the successor statement is not of interest, it will be omitted, as in $[stmt]^{\ell}$. The data interpretation of locations is denoted as memory accesses, i.e., $m_b[\ell] = n_b$ for $\ell \mapsto (\cdot, \cdot, n_b) \in \mathbf{CodeData}$.

Explicitly stating the address of the next statement in the code interpretation allows to deal with variable instruction length architectures such as x86, where the start of the next executed instruction depends on the length of the current instruction. Since a single assembly instruction can correspond to multiple IL statements, the bit length of locations may have to be extended to assign a unique address to each statement. Implementation details of the instruction translations are discussed in Section 5.1.3.

In an actual binary, **CodeData** is implicitly present by decoding the instruction at an address and translating it into the intermediate language. The finite partial mapping of locations to statements given through valid instruction decodings can be extended to a finite total mapping by setting the statements at all addresses that do not contain valid code to $\texttt{assert}\ 0_1$. Note that this assumes that hardware exceptions for invalid instructions are not caught; to achieve a more accurate model of the hardware behavior it is possible to fill the gaps with jumps to code that checks for an exception handler.

Control Flow Automata for IL Programs. Following the definition by Henzinger et al. [68], the *control flow automaton* (CFA) is a graph representation of a program, where nodes represent logical states and all statements are placed at the edges. This concept differs from the well-known control flow graph (CFG), a classical structure to reason about programs, in which nodes holding program statements are connected by edges denoting the possible control flow [1].

The CFA is a high level structure, in which code is clearly distinct from data. All control flow is explicit, since every edge has a definite start and end location. The advantage of this definition will become apparent in Chapter 3, which shows how jmp statements are resolved into multiple CFA edges labeled with assume statements. The definition of CFAs in [68] can be adapted to IL programs as follows:

Definition 2.1 (Control Flow Automaton) *A control flow automaton (CFA) is a tuple $\langle T, V, \mathbf{start}, E\rangle$, with a set of program locations $T \subseteq \mathbf{L}$, a set of registers V, an initial location $\mathbf{start} \in T$, and an edge relation $E \subseteq T \times \mathbf{Stmt} \times T$.*

2.6 Related Work

As mentioned above, the IL used in this dissertation is inspired by the semantics specification language (SSL) by Cifuentes and Sendall [38]. In particular, the instruction specifications implemented in Jakstab are based on SSL definitions for the Intel Pentium processor, which are part of the Boomerang decompiler project [21, 54]. The set of basic IL statements is largely analogous to SSL, except that SSL does not use explicit jumps but treats the program counter register as a regular register. The original SSL does not define statements and expressions for analysis (such as alloc and *nondet*), however, as it is primarily meant for binary translation between processor architectures [36].

The CodeSurfer/x86 project originally suffered from the problem that every analysis in principle had to implement an abstract transformer for each x86 assembly instruction, which in practice led to omitting large parts of the instruction set. Lim and Reps [88] address this problem with their own Transformer Specification Language (TSL). They conceptualize their approach as decomposing instructions into analysis primitives and specifying the abstract transformers in terms of the analysis primitives [87]. In their approach, analysis primitives basically take the place of IL statements and expressions. The conceptual difference between TSL and SSL mainly affects analysis implementations: Using SSL to translate a binary into IL creates an equivalent IL program, which is interpreted by an analysis. In TSL, the instruction specifications are C++ code that directly calls into a set of analysis-provided transformers. TSL thus avoids the overhead of creating an intermediate representation of assembly instructions, but its instruction specifications are tied to using C++ as implementation language.

A number of instruction specification languages have been described in the literature in the context of system design and compiler generation. The specification language λ-RTL by Ramsey and Davidson [116] is geared towards the development of modular compilers, such that instruction specifications can be reused for different languages. The language LISAS [43, 65] for describing instruction semantics is meant to augment circuit-level hardware description languages with "high level", i.e., instruction-level, information. Kästner [77] describes TDL, a very precise hardware specification language, for use in WCET analysis and compiler optimizations. The choice to base Jakstab's descriptions on SSL and to not use one of the other existing specification languages was decided by the availability of a fairly extensive x86 description.

Instruction specification languages are not limited to physical hardware. Eichberg et al.[53] presented a system which includes specifications of *bytecode* instructions and allows to specify generic bytecode analyses independent of a specific bytecode format such as Java bytecode or Microsoft's CIL. While the basic concept of abstractly representing multiple low level architectures is similar, bytecode contains types and structures and thus has different requirements for a specification language.

Dullien and Porst [51] introduce the intermediate language REIL for the purpose of reverse engineering. For the most part, REIL is a small subset of x86 assembly normalized to three address code. Besides basic arithmetic and bitwise operations, they add an *undef* statement that corresponds to the *nondet* assignment used in this dissertation. The advantage of their approach is that all statements use only basic operands and no complex expression types. The disadvantage, on the other hand, is that the lack of expressions requires more statements (they report on having 17 already without support for SIMD or floating point instructions). Furthermore, the absence of complex expressions prohibits code transformations such as forward expression substitution (see Section 5.3.6).

In the analysis of high level languages, problems similar to the richness of assembly language can arise if all flavors of syntactic sugar in different language implementations are to be supported. Therefore a number of intermediate languages exist for high level code as well. The C Intermediate Language [107] translates programs into a simpler subset of the C language and has become a popular front end for static analysis tools targeting C. Compilers have a long history of using intermediate languages for their code analysis, and several ILs exist that are designed to be independent of the high level source language. The LLVM project offers a common infrastructure for analyzing and compiling source languages [86], and is widely used in academia and industry. Similar to Microsoft Research's Phoenix infrastructure [97] or the GIMPLE language of GCC [58], it is primarily designed for compilation and relies on the availability of types and information about the program. Most analysis is carried out on the high level, typed intermediate language. The translation to machine code is performed near the end of the compilation process, and only some machine dependent optimizations are performed thereafter. A thorough analysis of low level code or a translation back to higher language levels does not lie within the focus of these frameworks.

Chapter 3

Control Flow Analysis for Low Level Programs

The IL is a direct translation from machine instructions, and both share a common problem that makes it hard to analyze machine code: The control flow is not explicit in the syntactic program representation, due to indirect jumps to computed target addresses. An essential step for analyzing binaries is thus to reconstruct the control flow automaton, i.e., to statically determine all possible targets of dynamically computed indirect jumps. This chapter presents a general framework to construct an overapproximation of the CFA of an IL program by effectively combining control and data flow analysis, akin to control flow analysis in functional programming languages [76, 126]. The framework is based on the general concept of abstract interpretation [45] and not fixed to a particular abstract domain for data flow analysis. It does not require additional information besides the actual statements and is still able to compute a sound and precise overapproximation of a program's control flow.

3.1 Overview

Data flow analysis statically calculates information about the program variables from a given program representation. For instance, *Constant Propagation* is a simple data flow analysis that calculates for each program location and for each program variable the constant value of the variable at this location, or the special value $\top$ if the variable cannot be shown to be constant by the analysis. The analysis determines the program locations to which the effects of an assignment can flow using the control flow graph of the program. Earlier work [7, 50, 78, 81, 133] has shown that data flow analysis can be used to augment the results of disassembly. No conclusive answer was given, however, how states with unresolved control flow successors should be handled during data flow analysis such that the resulting control flow graph is optimal. Furthermore, it was unclear whether updating the control flow graph could render previous data flow information invalid, which would require backtracking and could cause the analysis to diverge.

This chapter will demonstrate how to successfully design a program analysis that reconstructs the control flow of a low level program, and prove that the notion of a “chicken and egg problem” in combining data flow analysis with disassembly is overly pessimistic. The approach presented here is based on the idea of executing data flow analysis and branch resolution simultaneously. A data flow problem is characterized by a constraint system derived from an over-approximation of the program semantics. The solution to a data flow problem is calculated by iteratively applying the constraints until a fixed point is reached. A constraint system for data flow analysis has the form

$$D(\ell) \sqsupseteq \bigsqcup_{(\ell',stmt,\ell)\in G} \widehat{\mathbf{post}}[\![stmt]\!](D(\ell')) \sqcup \iota^{\ell},$$

where $\widehat{\mathbf{post}}$ is the abstract post operator (also called transfer function, state-, or predicate transformer), D records data flow information for locations $\ell \in \mathbf{L}$, and ι^{ℓ} is the initial data flow information for ℓ. The constraint system specifies that the data flow information for every location ℓ is the combination of the data flow information of its predecessors ℓ', updated according to the semantics of the statements leading to ℓ (forward analysis). The predecessor relationship is encoded in the control flow automaton G, which is unavailable in binaries, however. Indeed, determining G is the whole purpose of data flow-assisted control flow reconstruction.

The intuition of the approach presented here is that the edge relation grows during the fixed point iteration until a simultaneous least fixed point of both data and control flow is reached. A **resolve** operator is responsible for growing the edge relation and uses data flow information to calculate branch targets of instructions. The combined analysis ensures that:

- The quality of the fixed point, and thus of the reconstructed CFA, does not depend on the order in which the constraints are applied.
- The fixed point of control and data flow is an overapproximation of the concrete program semantics.

The basic idea of the framework is to translate statements into edges ($\mathbf{L} \times \mathbf{Stmt} \times \mathbf{L}$) of the control flow automaton (CFA edges). The edges overapproximate the concrete control flow of the program, eliminating any indirect jumps. In particular, every guarded jump $[\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]^{\ell}_{\ell'}$ is transformed into a set of edges labeled with assume statements. If the condition e_1 is false, the set contains only the fall-through edge to ℓ', labeled with assume $(e_1 = 0_1)$. Otherwise, the set will also contain edges to all addresses that are possible values of the target expression e_2 in the current data flow state. Each of these edges will be labeled with statements assuming both the condition $e_1 \neq 0_1$ and the fact that e_2 evaluates to its target location. The encoding of branch conditions and jump targets as assume statements is the key feature why this approach produces the most precise CFA with respect to the precision of the data flow analysis.

A data flow analysis used to instantiate this framework needs to supply only implementations of the $\widehat{\mathbf{post}}$ (for statements other than jmp) and $\widehat{\mathbf{eval}}$ operators and does not need to deal specifically with indirect jumps. To this end, the framework extends the provided abstract domain with a set of edges that represents those parts of the final CFA that have been explored so far and an implementation of $\widehat{\mathbf{post}}$ for jumps. Section 3.3 defines the notion of the concrete CFA for

IL programs based on their concrete semantics. The **resolve** operator, which is constructed using conditions imposed on the provided abstract domain, calculates targets for low level branch instructions. Using this operator, the analysis is able to safely overapproximate the concrete CFA (Section 3.4).

The classical worklist algorithm met in program analysis is extended such that it supports control flow reconstruction with the aid of data flow analysis under very general assumptions. The algorithm overcomes the "chicken and egg" problem of how to bootstrap the analysis process by computing the a priori unknown edges on the fly with the help of the **resolve** operator. It is proven that the algorithm always returns the most precise overapproximation of the program's actual control flow automaton with respect to the precision of the provided abstract domain used by the data flow analysis (Section 3.5).

3.2 A Worked Example

Let us first illustrate the proposed approach with a small example. Consider the IL program shown on the left of Figure 3.1 that performs some address arithmetic and contains an indirect jump to the computed target address. The abstract domain for the analysis is chosen to model for each program location a set of up to 5 values for the single program variable x. The top element $\top$ represents sets larger than 5, and the bottom element $\perp$ denotes that a location has not been reached by the analysis yet. The entry point of the program is at address 0, and the data flow information for the program, denoted as D, is initialized to $D(0) = \top$ and $D(\ell) = \perp$ for $\ell \neq 0$. The initial control flow automaton $G = \emptyset$ contains no edges.

The steps of the worklist algorithm that will be outlined in this chapter are shown in Table 3.1. Control flow reconstruction begins by calling the **resolve** operator to generate the first edges from the only location where the data flow information is not $\perp$, the entry point. The call yields two outgoing edges, since nothing is known about x (step 1). The jump condition is transformed into as-

ℓ	Statement	$D(\ell)$
0:	if $(x = 0)$ jmp 13	$\top$
5:	$x := 1$	$\top$
10:	if 1 jmp 21	$\{1\}$
12:	halt	$\{12\}$
13:	$x := 24$	$\{0\}$
18:	$x := x - 5$	$\{18, 24\}$
21:	$x := x - 1$	$\{1, 13, 19\}$
24:	if 1 jmp x	$\{0, 12, 18\}$

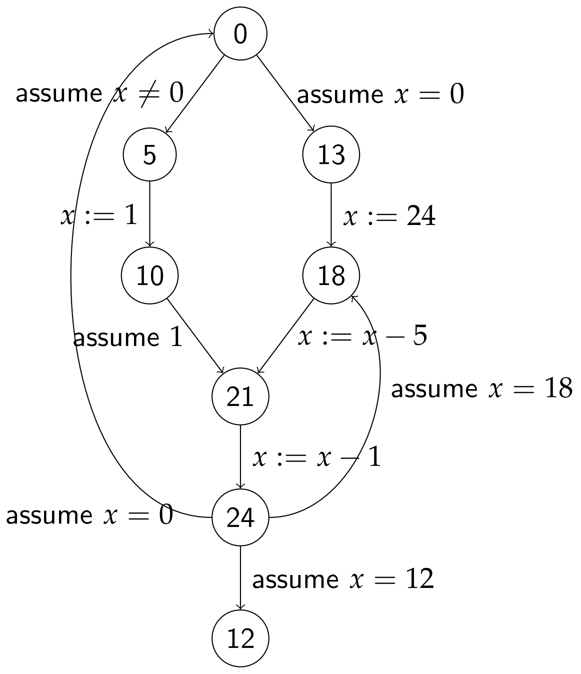


Figure 3.1: Control flow reconstruction example: IL program, final data flow information, and reconstructed control flow automaton.

sumptions on whether the jump condition was true or false. The algorithm now computes the transfer function of the data flow analysis for these two edges and updates $D(5)$ and $D(13)$ accordingly (steps 2 and 3). Note that the abstract domain cannot precisely express the fact that $x \neq 0$.

The analysis now follows down both branches and updates D until the common postdominator (the first common descendant of both branches), i.e., location 21 (steps 4-11). In this abstract domain, the data flow information from both branches is joined in location 21, yielding the set $\{1, 19\}$ (step 11) and subsequently $\{0, 18\}$ at the location of the indirect jump (step 12). These two values cause two new assume edges encoding the fact that the jump target evaluated to 0 or 18 (step 13). The edge to location 0 does not lead to new data flow information, since $D(0) = \top \supseteq \{0\}$. The edge to location 18, however, adds an additional value to $D(18)$, which in turn leads to new data flow information in

Step	**Data flow update**	**Control flow update**
Init	$D(0) = \top$	$G = \emptyset$
1		$(0, \mathsf{assume}\ x \neq 0, 5), (0, \mathsf{assume}\ x = 0, 13)$
2	$D(5) = \top$	
3	$D(13) = \{0\}$	
4		$(5, x := 1, 10)$
5	$D(10) = \{1\}$	
6		$(13, x := 24, 18)$
7	$D(18) = \{24\}$	
8		$(10, \mathsf{assume}\ 1, 21)$
9	$D(21) = \{1\}$	
10		$(18, x := x - 5, 21)$
11	$D(21) = \{1, 19\}$	
12	$D(24) = \{0, 18\}$	
13		$(24, \mathsf{assume}\ x = 0, 0), (24, \mathsf{assume}\ x = 18, 18)$
14	$D(18) = \{18, 24\}$	
15	$D(21) = \{1, 13, 19\}$	
16	$D(24) = \{0, 12, 18\}$	
17		$(24, \mathsf{assume}\ x = 12, 12)$
18	$D(12) = \{12\}$	
19	Fixpoint	Fixpoint

Table 3.1: Example run of the worklist control flow reconstruction algorithm.

the successor locations (steps 14-16). Thus, a new possible target value is propagated to the location of the indirect jump, producing a new edge to location 12 (step 17). Finally, $D(12)$ at the location of the halt statement is updated, and the analysis has reached a fixpoint where neither data flow nor control flow can be updated anymore. The final data flow information and the final control flow automaton are shown on the right of Figure 3.1.

3.3 Control Flow Semantics

For reconstructing the control flow of an IL program, it suffices to only view the basic set of statements introduced in Section 2.2.2. The target control flow automaton can be built using an even smaller set of statements $\mathbf{Stmt}^\#$, which consists only of register or memory assignments and assume statements but does not contain guarded jump statements. The intuition is that the assume statements are generated from resolving a guarded jump statement and that each successor address ℓ' represents the resolved target address of the jump. The condition being assumed encodes (i) whether the statement represents the true or false branch of the guarded jump and (ii) for true branches, the fact that the jump's target expression evaluated to the target location of the edge.

The operator $\mathbf{post} : \mathbf{Stmt}^\# \to \mathbf{2}^{\mathbf{State}} \to \mathbf{2}^{\mathbf{State}}$ is overloaded to work on statements of the derived language and *sets* of states $S \subseteq \mathbf{State}$:

$$\begin{aligned}
\mathbf{post}[\![[v := e]^{\ell}_{\ell'}]\!](S) \quad &:= \{\mathbf{post}[\![[v := e]^{\ell}_{\ell'}]\!](s) \mid s \in S\},\\
\mathbf{post}[\![[m[e_1] := e_2]^{\ell}_{\ell'}]\!](S) \quad &:= \{\mathbf{post}[\![[m[e_1] := e_2]^{\ell}_{\ell'}]\!](s) \mid s \in S\},\\
\mathbf{post}[\![[\mathsf{assume}\ e]^{\ell}_{\ell'}]\!](S) \quad &:= \{s[pc \mapsto \ell'] \mid \mathbf{eval}[\![e]\!](s) \neq 0_1, s \in S\}.
\end{aligned}$$

Note that the definition of the **post** operator over sets makes use of the **post** operator for single elements in the case of assignments. $\mathbf{Stmt}^\#$ and the definition of the transfer function **post** will be used for stating the conditions required from the abstract domain for the control flow reconstruction in Section 3.4.

For the sake of continuity, the framework is described here in terms of the IL introduced in Chapter 2; it is possible, however, to use a slightly more generalized low level language that operates on integers instead of bit-vectors, as long as every program still has a fixed finite representation, i.e., the set of program locations **L** is finite [83].

Definition 3.1 (Trace) *A trace σ of a program is a finite sequence of states $(s_i)_{0\leq i\leq n}$, such that $s_0(pc) = \mathbf{start}$, $stmt \in \mathbf{Stmt}$ is not* halt *for all $[stmt]^{s_i(pc)}$ with $0 \leq i < n$, and $s_{i+1} = \mathbf{post}[\![stmt]\!](s_i)$ for all $[stmt]^{s_i(pc)}$ with $0 \leq i < n$.*

The register and store valuations for state s_0 can be freely defined by the abstract domain. The set of all traces of a program is denoted by **Traces**. Further, the program counter of all states in all traces is assumed to only map into the finite set of locations **L**, as every program has a fixed finite representation.

The goal is to reconstruct the control flow of a program, therefore the *concrete control flow automaton* has to be defined as the desired property to be approximated by the analysis. The definition of concrete CFAs of IL programs (Definition 3.3) is based on the definition of traces and uses labeled edges. The set of labeled edges **Edge** is defined as $\mathbf{L} \times \mathbf{Stmt}^{\#} \times \mathbf{L}$. Initially, the concrete CFA for a single trace is defined:

Definition 3.2 (Concrete TCFA) *Given a trace $\sigma = (s_i)_{0\leq i\leq n}$, the concrete trace control flow automaton (TCFA) of σ is*

$$\begin{aligned}
TCFA(\sigma) = \{&(s_i(pc), stmt, s_{i+1}(pc)) \mid \\
&\qquad 0 \leq i < n \text{ with } [stmt]^{s_i(pc)}, \text{ where } stmt \text{ is } v := e \text{ or } m[e_1] := e_2\} \\
\cup \{&(s_i(pc), \mathsf{assume}\ (e_1 = 0_1), s_{i+1}(pc)) \mid \\
&\qquad 0 \leq i < n \text{ with } [\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]^{s_i(pc)} \text{ and } \mathbf{eval}[\![e_1]\!](s_i) = 0_1\} \\
\cup \{&(s_i(pc), \mathsf{assume}\ (e_1 \neq 0_1 \wedge e_2 = s_{i+1}(pc)), s_{i+1}(pc)) \mid \\
&\qquad 0 \leq i < n \text{ with } [\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]^{s_i(pc)} \text{ and } \mathbf{eval}[\![e_1]\!](s_i) \neq 0_1\}.
\end{aligned}$$

Now the concrete CFA for a full program can be defined as the combination of all TCFAs:

Definition 3.3 (Concrete CFA) *The concrete control flow automaton is the union of the TCFAs of all traces:*

$$CFA = \bigcup_{\sigma \in \mathbf{Traces}} TCFA(\sigma).$$

Note that by this definition, the CFA of a program is a semantic property rather than a syntactic one, since it depends on the possible concrete executions. Unlike control flow graphs built purely from a syntactic program representation, it does not contain infeasible edges. With respect to the forward collecting semantics of reachable states, the concrete CFA of an IL program (with an initial state including all static data) is equivalent to the description of the program by **CodeData** (see Section 2.5), a mapping from addresses to statements that includes guarded jumps. All possible executions are represented by the union of traces, and the assume statements represent the outcome of conditions and target expressions for guarded jumps.

3.4 Control Flow Reconstruction by Abstract Interpretation

The framework for control flow reconstruction is parameterized by an abstract domain that provides information about the data state of the program. The framework accepts abstract domains $(A, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$, where

- $(A, \bot, \top, \sqcap, \sqcup, \sqsubseteq)$ is a complete lattice, with the set of lattice elements A, the bottom element $\bot$, the top element $\top$, the meet $\sqcap$, the join $\sqcup$, and the partial order $\sqsubseteq$,
- the *concretization* function $\gamma : A \to \mathbf{2}^{\mathbf{State}}$ from abstract lattice elements to sets of concrete states is monotone, i.e.,

$$a_1 \sqsubseteq a_2 \Rightarrow \gamma(a_1) \subseteq \gamma(a_2) \quad \text{for all } a_1, a_2 \in A,$$

and maps the least element to the empty set, i.e., $\gamma(\bot) = \emptyset$,

- the *abstract post operator* $\widehat{\mathbf{post}} : \mathbf{Stmt}^{\#} \to A \to A$ overapproximates the concrete transfer function **post**, i.e.,

$$\mathbf{post}[\![stmt]\!](\gamma(a)) \subseteq \gamma(\widehat{\mathbf{post}}[\![stmt]\!](a)) \text{ for all } stmt \in \mathbf{Stmt}^{\#}, a \in A, \text{ and}$$

- the *abstract evaluation* function $\widehat{\mathbf{eval}} : \mathbf{Exp} \to A \to \mathbf{2}^{\mathbb{Z}}$ overapproximates the concrete evaluation function, i.e.,

$$\bigcup_{s \in \gamma(a)} \mathbf{eval}[\![e]\!](s) \subseteq \widehat{\mathbf{eval}}[\![e]\!](a) \text{ for all } e \in \mathbf{Exp}, a \in A.$$

  Note that by this definition, the abstract evaluation function returns sets of integers and not an abstract value type.

The next sections define a control flow analysis based on a given abstract domain $(A, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$. The control flow analysis works on a *Cartesian abstract domain* $D : \mathbf{L} \to A$, which records data flow facts for individual program locations, and a *set of control flow edges* $G \subseteq \mathbf{Edge}$, which stores the edges that have been explored so far.

3.4.1 The Resolve Operator

A control flow analysis must have the ability to detect the (possibly overapproximated) set of targets of guarded jumps based on the knowledge it acquires. This is accomplished by the operator **resolve** : $\mathbf{L} \to A \to \mathbf{2}^{\mathbf{Edge}}$, which is defined using the functions available in the abstract domain. For a given location ℓ and an abstract data flow lattice element a, **resolve** returns a set of edges of the control flow automaton. If a is the least element $\bot$, the location ℓ has not been reached by the abstract interpretation yet, therefore no edge needs to be created and the empty set is returned. Otherwise, **resolve** labels fall-through edges with their respective source statements, or it calculates the targets of guarded jumps based on the information gained from the data flow lattice element a and labels the determined edges with their respective conditions. The operator is formally defined as:

$$\mathbf{resolve}_\ell(a) :=$$

$$:= \begin{cases} \emptyset & \text{if } a = \bot \text{ or } [stmt]^\ell_{\ell'} \text{ is } [\mathsf{halt}]^\ell_{\ell'} \\ \{(\ell, stmt, \ell')\} & \text{if } a \neq \bot \text{ and } ([v := e]^\ell_{\ell'} \text{ or } [m[e_1] := e_2]^\ell_{\ell'}) \\ \{(\ell, \mathsf{assume}\ (e_1 \neq 0_1 \wedge e_2 = \ell''), \ell'') \mid & \text{if } a \neq \bot \text{ and } [\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]^\ell_{\ell'} \\ \quad \ell'' \in \widehat{\mathbf{eval}}[\![e_2]\!]\big(\widehat{\mathbf{post}}[\![\mathsf{assume}\ (e_1 \neq 0_1)]\!](a)\big)\big\} & \\ \cup\{(\ell, \mathsf{assume}\ (e_1 = 0_1), \ell')\} & \end{cases}$$

The crucial part in this definition is the last case, where the abstract $\widehat{\mathbf{post}}$ and the abstract $\widehat{\mathbf{eval}}$ are used to calculate possible jump targets. The set of jump targets is determined by abstractly evaluating the target expression in those states which satisfy the jump condition. Note that evaluating the target expression in abstract state a instead of the restricted state $\widehat{\mathbf{post}}[\![\mathsf{assume}\ (e_1 \neq 0_1)]\!](a)$ would include states that do not pass the jump condition, which would be a legal but unnecessarily imprecise overapproximation. This definition uses operators defined by the abstract domain, therefore the precision of the control flow analysis is influenced by the precision of the abstract domain.

3.4.2 A Constraint System for Control Flow Automata

With the definition of the **resolve** operator, it is possible to state a system of constraints such that all solutions of these constraints are solutions to the control flow analysis. The first component is the Cartesian abstract domain $D : \mathbf{L} \to A$, which maps addresses to elements of the abstract domain. The idea is that D

captures the data flow facts derived from the program. The second component is the set of edges $G \subseteq \mathbf{Edge}$ which stores the edges produced by the **resolve** operator. Finally, an initial abstract element $\iota^\ell \in A$ is defined for every location $\ell \in \mathbf{L}$. With these components, the joint system of constraints becomes:

$$G \supseteq \bigcup_{\ell \in \mathbf{L}} \mathbf{resolve}_\ell(D(\ell)) \tag{3.1}$$

$$D(\ell) \sqsupseteq \bigsqcup_{(\ell', stmt, \ell) \in G} \widehat{\mathbf{post}}[\![stmt]\!](D(\ell')) \sqcup \iota^\ell \tag{3.2}$$

Note that the initial abstract elements ι^ℓ are outside the scope of the join over all predecessors; addresses with no predecessor statements thus keep their initial abstract element. By this definition, G does not only store the a priori unknown targets of the guarded jumps, but also the conditions (assume statements) which have to be satisfied to reach them. These conditions can be used by the abstract $\widehat{\mathbf{post}}$ to propagate precise information.

For a unified view of the analysis problem, the system of constraints (3.1) and (3.2) can be combined into a single function

$$F : \left((\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}}\right) \to \left((\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}}\right),$$

which is defined as

$$F(D, G) \mapsto (D', G'),$$

where

$$G' = \bigcup_{\ell \in \mathbf{L}} \mathbf{resolve}_\ell(D(\ell)),$$
$$D'(\ell) = \bigsqcup_{(\ell', stmt, \ell) \in G} \widehat{\mathbf{post}}[\![stmt]\!](D(\ell')) \sqcup \iota^\ell.$$

The connection between constraints (3.1) and (3.2) and control flow analysis is stated in the following theorem, where correctness depends on $\iota^{\mathbf{start}} \in \mathbf{L}$:

Theorem 3.4 *Given an IL program and a trace $\sigma = (s_i)_{0 \le i \le n}$, such that $s_0(pc) =$ **start** and $s_0 \in \gamma(\iota^{\mathbf{start}})$, every solution (D, G) of the constraints (3.1) and (3.2) satisfies $s_n \in \gamma(D(s_n(pc)))$ and $TCFA(\sigma) \subseteq G$.*

The proof is a straightforward induction on the length of traces using the properties required from the abstract domain:

Proof Let (D, G) be a solution to the system of constraints (3.1) and (3.2). By induction on n, we show that $s_n \in \gamma(D(s_n(pc)))$ and $TCFA(\sigma) \subseteq G$ for every trace $\sigma = (s_i)_{0 \le i \le n}$ with $s_0 \in \gamma(\iota^{\mathbf{start}})$. For the induction basis $n = 0$, we consider the trace $(t_i)_{0 \le i \le 0}$ of length 1 with $t_0 \in \gamma(\iota^{\mathbf{start}})$. We have

$$t_0 \in \gamma(\iota^{\mathbf{start}}) \subseteq \gamma\left(\left(\bigsqcup_{(\ell', stmt, \mathbf{start}) \in G} \widehat{\mathbf{post}}[\![stmt]\!](D(\ell'))\right) \sqcup \iota^{\mathbf{start}}\right) \subseteq \gamma(D(\mathbf{start})),$$

where the element relationship holds by assumption. The first inclusion holds due to the monotonicity of γ. The second inclusion holds because (D, G) is a solution to the system of constraints (3.1) and (3.2) and because γ is monotone. Furthermore we have $TCFA((t_0)) = \emptyset \subseteq G$.

For the inductive step $n \mapsto n + 1$, we assume that for all traces $\sigma = (s_i)_{0 \le i \le n}$ of length $(n + 1)$ with $s_0 \subseteq \gamma(\iota^{\mathbf{start}})$ the inclusions $s_n \subseteq \gamma(D(s_n(pc)))$ and $TCFA(\sigma) \subseteq G$ hold. We consider the trace $(t_i)_{0 \le i \le n+1}$ of length $(n + 2)$ with $t_0 \in \gamma(\iota^{\mathbf{start}})$ and proceed by case distinction on the statement *stmt*, with $[stmt]^{t_n(pc)}$ being part of the program.

- *stmt* is halt. The trace $(t_i)_{0 \le i \le n+1}$ has a state t_{n+1}, making this case impossible by the definition of a trace. Note that halt instructions are themselves never part of a trace, but instead cause the trace to end.

- *stmt* is $v := e$ or $m[e_1] := e_2$. By induction assumption we know for the trace $(t_i)_{0 \le i \le n}$ that $t_n \in \gamma(D(t_n(pc)))$. Therefore we have that $D(t_n(pc)) \neq \bot$, since $\gamma(\bot) = \emptyset$ (t_n is not contained in the empty set). It follows that

$$\begin{aligned}(t_n(pc), stmt, t_{n+1}(pc)) &\in \mathbf{resolve}_{t_n(pc)}(D(t_n(pc))) \\ &\subseteq \bigcup_{\ell \in \mathbf{L}} \mathbf{resolve}_{\ell}(D(\ell)) \\ &\subseteq G \qquad (3.3)\end{aligned}$$

where we have the element relationship from the definition of **resolve**, the first inclusion holds trivially, and the second because (D, G) is a solution of constraint system (3.1) and (3.2). As we have $TCFA((t_i)_{0 \le i \le n}) \subseteq G$ by induction assumption, we now have $TCFA((t_i)_{0 \le i \le n+1}) \subseteq G$.

Furthermore, we deduce

$$\begin{aligned} t_{n+1} &= \mathbf{post}[\![stmt]\!](t_n) \\ &\in \mathbf{post}[\![stmt]\!]\left(\gamma(D(t_n))\right) \\ &\subseteq \gamma\left(\widehat{\mathbf{post}}[\![stmt]\!](D(t_n))\right) \\ &\subseteq \gamma\left(\bigsqcup_{(\ell', stmt, t_{n+1}(pc)) \in G} \widehat{\mathbf{post}}[\![stmt]\!](D(\ell'))\right) \\ &\subseteq \gamma(D(t_{n+1})), \end{aligned}$$

where the equality holds by the definition of a trace, the element relationship holds by induction assumption. The first inclusion holds as the abstract $\widehat{\mathbf{post}}$ operator is an overapproximation of the concrete **post** operator. The second inclusion holds because we have $(t_n(pc), stmt, t_{n+1}(pc)) \in G$ by (3.3) and γ is monotone. The third inclusion holds due to the facts that (D, G) is a solution of the system of constraints (3.1) and (3.2) and that γ is a monotone function.

- *stmt* is if e_1 jmp e_2. Again, by induction assumption we have for the trace $(t_i)_{0 \le i \le n}$ that $t_n \in \gamma(D(t_n(pc)))$. Therefore we know that $D(t_n(pc)) \neq \bot$, since $\gamma(\bot) = \emptyset$. Now we proceed by case distinction on the value of $\mathbf{eval}[\![e_1]\!](t_n)$.

– $\mathbf{eval}[\![e_1]\!](t_n) = 0_1$. We infer

$$\begin{aligned}(t_n(pc), \mathsf{assume}\ (e_1 = 0_1), t_{n+1}(pc)) &\in \mathbf{resolve}_{t_n(pc)}(D(t_n(pc))) \\ &\subseteq \bigcup_{\ell \in \mathbf{L}} \mathbf{resolve}_{\ell}(D(\ell)) \\ &\subseteq G, \end{aligned} \tag{3.4}$$

where we have the element relationship by the definition of **resolve**. The first inclusion holds trivially. The second inclusion follows from (D, G) being a solution of the system of constraints (3.1) and (3.2). By induction assumption we know that $\mathit{TCFA}((t_i)_{0 \le i \le n}) \subseteq G$, so we now have $\mathit{TCFA}((t_i)_{0 \le i \le n+1}) \subseteq G$.

We deduce

$$\begin{aligned} t_{n+1} &= \mathbf{post}[\![\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]\!](t_n) \\ &\in \mathbf{post}[\![\mathsf{assume}\ (e_1 = 0_1)]\!](\{t_n\}) \\ &\subseteq \mathbf{post}[\![\mathsf{assume}\ (e_1 = 0_1)]\!](\gamma(D(t_n))) \\ &\subseteq \gamma\left(\widehat{\mathbf{post}}[\![\mathsf{assume}\ (e_1 = 0_1)]\!](D(t_n))\right) \\ &\subseteq \gamma\left(\bigsqcup_{(\ell', \mathit{stmt}, t_{n+1}(pc)) \in G} \widehat{\mathbf{post}}[\![\mathit{stmt}]\!](D(\ell'))\right) \\ &\subseteq \gamma(D(t_{n+1})), \end{aligned}$$

where we have the initial equality from the definition of a trace. We have the element relationship from the definition of the **post** operator and from the case assumption $\mathbf{eval}[\![e_1]\!](t_n) = 0_1$. We know the first inclusion from the induction assumption. The second inclusion holds as the abstract $\widehat{\mathbf{post}}$ operator is an overapproximation of the concrete **post** operator. The third inclusion holds because we have $(t_n(pc), \mathsf{assume}\ (e_1 = 0_1), t_{n+1}(pc)) \in G$ by (3.4) and because γ is a monotone function. The fourth inclusion holds as (D, G) is a solution of the system of constraints (3.1) and (3.2) and because γ is a monotone function.

This completes treatment of the fall-through case for jumps.

– $\mathbf{eval}[\![e_1]\!](t_n) \neq 0_1$. We infer

$$\begin{aligned} t_{n+1}(pc) &= \mathbf{eval}[\![e_2]\!](t_n) \\ &\in \mathbf{eval}[\![e_2]\!]\Big(\mathbf{post}[\![\mathsf{assume}\ (e_1 \neq 0_1)]\!](\{t_n\})\Big) \\ &\subseteq \mathbf{eval}[\![e_2]\!]\Big(\mathbf{post}[\![\mathsf{assume}\ (e_1 \neq 0_1)]\!](\gamma(D(t_n)))\Big) \\ &\subseteq \mathbf{eval}[\![e_2]\!]\left(\gamma\left(\widehat{\mathbf{post}}[\![\mathsf{assume}\ (e_1 \neq 0_1)]\!](D(t_n))\right)\right) \\ &\subseteq \widehat{\mathbf{eval}}[\![e_2]\!]\left(\widehat{\mathbf{post}}[\![\mathsf{assume}\ (e_1 \neq 0)]\!](D(t_n))\right), \end{aligned} \tag{3.5}$$

where we have the equality by the definition of **post**. The element relationship follows from the case assumption $\mathbf{eval}[\![e_1]\!](t_n) \neq 0_1$, since $\mathbf{post}[\![\mathsf{assume}\ (e_1 \neq 0_1)]\!](\{t_n\})$ is equal to the unchanged singleton set $\{t_n\}$ in this case. The first inclusion holds by induction assumption. The second inclusion holds as, by definition of the abstract domain, the abstract $\widehat{\mathbf{post}}$ operator is an overapproximation of the concrete **post** operator. The third inclusion holds as the abstract $\widehat{\mathbf{eval}}$ is an overapproximation of the concrete **eval**, again by the conditions imposed on the abstract domain.

Furthermore, we have $t_{n+1}(pc) \in \mathbf{L}$ from the definition of traces. We therefore deduce

$$\begin{aligned} (t_n(pc), \mathsf{assume}\ (e_1 \neq 0_1), t_{n+1}(pc)) &\in \mathbf{resolve}_{t_n(pc)}(D(t_n(pc))) \\ &\subseteq \bigcup_{\ell \in \mathbf{L}} \mathbf{resolve}_\ell(D(\ell)) \\ &\subseteq G, \end{aligned} \tag{3.6}$$

where we have the element relationship from (3.5) and from the fact that $t_{n+1} \in \mathbf{L}$. The first inclusion holds trivially. The second inclusion holds because (D, G) is a solution of the system of constraints (3.1) and (3.2). As we have $TCFA((t_i)_{0 \le i \le n}) \subseteq G$ by induction assumption, we now have $TCFA((t_i)_{0 \le i \le n+1}) \subseteq G$.

Finally, we deduce

$$\begin{aligned}
t_{n+1} &= \mathbf{post}[\![\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]\!](t_n) \\
&\in \mathbf{post}[\![\mathsf{assume}\ (e_1 \neq 0_1)]\!]\{(t_n)\} \\
&\subseteq \mathbf{post}[\![\mathsf{assume}\ (e_1 \neq 0_1)]\!](\gamma(D(t_n(pc)))) \\
&\subseteq \gamma\left(\widehat{\mathbf{post}}[\![\mathsf{assume}\ (e_1 \neq 0_1)]\!](D(t_n))\right) \\
&\subseteq \gamma\left(\bigsqcup_{(\ell', stmt, t_{n+1}(pc)) \in G} \widehat{\mathbf{post}}[\![stmt]\!](D(\ell'))\right) \\
&\subseteq \gamma(D(t_{n+1})),
\end{aligned}$$

where we have the equality from the definition of a trace. We have the element relationship from the definition of **post**, and from the case assumption $\mathbf{eval}[\![e_1]\!](t_n) \neq 0_1$. The first inclusion holds by induction assumption. The second inclusion holds as the abstract $\widehat{\mathbf{post}}$ is an overapproximation of the concrete **post**. The third inclusion holds because we have $(t_n(pc), \mathsf{assume}\ (e_1 \neq 0_1), t_{n+1}(pc)) \in G$ by (3.6) and because γ is a monotone function. The fourth inclusion holds as (D, G) is a solution of constraint system (3.1) and (3.2) and because γ is a monotone function. □

From Theorem 3.4, we immediately obtain:

Corollary 3.5 *Given an IL program and a solution (D, G) of the constraints (3.1) and (3.2), where $\{s \in \mathbf{State} \mid s(pc) = \mathbf{start}\} \subseteq \gamma(\iota^{\mathsf{start}})$, G is a superset of the control flow automaton.*

The Cartesian abstract domain $\mathbf{L} \to A$, equipped with pointwise ordering, i.e., $D_1 \sqsubseteq D_2 :\Leftrightarrow \forall \ell \in \mathbf{L}.\, D_1(\ell) \sqsubseteq D_2(\ell)$, is a complete lattice because A is a complete lattice. The power set $\mathbf{2^{Edge}}$ ordered by the subset relation $\subseteq$ is a complete lattice. The product lattice $(\mathbf{L} \to A) \times \mathbf{2^{Edge}}$, equipped with pointwise ordering, i.e., $(D_1, G_1) \sqsubseteq (D_2, G_2) :\Leftrightarrow D_1 \sqsubseteq D_2 \wedge G_1 \subseteq G_2$, is complete as both $\mathbf{L} \to A$ and $\mathbf{2^{Edge}}$ are complete. Evidently, F is a monotone function on $(\mathbf{L} \to A) \times \mathbf{2^{Edge}}$. As $(\mathbf{L} \to A) \times \mathbf{2^{Edge}}$ is a complete lattice, the existence of a least fixed point μ of the function F follows from the Knaster-Tarski fixed point theorem [131]. Therefore, we immediately obtain:

Proposition 3.6 *The combined control and data flow problem, i.e., the system of constraints (3.1) and (3.2), always has a unique best solution.*

3.5 Algorithms for Control Flow Reconstruction

For the purpose of algorithm design this section focuses on abstract domains A satisfying the ascending chain condition (ACC) and presents two control flow reconstruction algorithms. The first algorithm (Figure 3.2) is generic and gives an answer to the "chicken and egg" problem as it computes a sound overapproximation of the CFA by an intertwined control and data flow analysis. Note that the order in which the control flow reconstruction is done may only affect efficiency but not precision. The second algorithm (Figure 3.3) is an extension of the classical worklist algorithm and is geared towards practical implementation.

3.5.1 Generic Fixed Point Algorithm

The generic algorithm shown in Figure 3.2 maintains a Cartesian abstract domain $D : \mathbf{L} \to A$ and a set of edges $G \subseteq \mathbf{Edge}$. $D(\ell)$ is initialized by $\iota^{\mathbf{start}}$ for $\ell = \mathbf{start}$ (line 4) and by $\perp$ for $\ell \neq \mathbf{start}$ (line 3). As the algorithm does not know anything about the control flow of the program yet, it starts with G as the empty set (line 5). The algorithm iterates its main loop as long as it can find an

```
Input: an IL program, its set of addresses $\mathbf{L}$ including $\mathbf{start}$, and the
       abstract domain $(A, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$ together with an
       initial value $\iota^{\mathbf{start}}$
Output: a control flow automaton
begin
    forall the $\ell \in \mathbf{L} \setminus \{\mathbf{start}\}$ do
        $D(\ell) := \bot$;
    $D(\mathbf{start}) := \iota^{\mathbf{start}}$;
    $G := \emptyset$;
    while true do
        $Choices := \emptyset$;
        if $\exists (\ell', stmt, \ell) \in G.\, \widehat{\mathbf{post}}[\![stmt]\!](D(\ell')) \not\sqsubseteq D(\ell)$ then
            $Choices := \{\text{do\_p}\}$;
        if $\exists \ell \in \mathbf{L}.\, \mathbf{resolve}_\ell(D(\ell)) \not\subseteq G$ then
            $Choices := Choices \cup \{\text{do\_r}\}$;
        if $\exists u \in Choices$ then
            choose $u \in Choices$ /* non-deterministic choice */
            switch $u$ do
                case do_p
                    choose $(\ell', stmt, \ell) \in G$ where $\widehat{\mathbf{post}}[\![stmt]\!](D(\ell')) \not\sqsubseteq D(\ell)$;
                    $D(\ell) := \widehat{\mathbf{post}}[\![stmt]\!](D(\ell')) \sqcup D(\ell)$;
                case do_r
                    choose $\ell \in \mathbf{L}$ where $\mathbf{resolve}_\ell(D(\ell)) \not\subseteq G$;
                    $G := \mathbf{resolve}_\ell(D(\ell)) \cup G$;
        else
            return G;
end
```

Figure 3.2: Generic Control Flow Reconstruction Algorithm.

unsatisfied inequality (lines 8 and 10). Thus the algorithm essentially searches for violations of constraints (3.1) and (3.2). If the generic algorithm finds at least one not yet satisfied inequality, it non-deterministically picks a single unsatisfied inequality and updates it (lines 13 to 20).

The correctness of the generic algorithm can now be established for abstract domains A that satisfy the ascending chain condition:

Theorem 3.7 *Given an IL program, where $\{s \in \textbf{State} \mid s(pc) = \textbf{start}\} \subseteq \gamma(\iota^{\textbf{start}})$, the generic control flow reconstruction algorithm (as defined in Figure 3.2) computes a sound overapproximation of the CFA and terminates in finite time. Furthermore it returns the most precise result with respect to the precision of the abstract domain A regardless of the non-deterministic choices made in line 13.*

The algorithm terminates because $(\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}'}$ satisfies the ascending chain condition, where $\mathbf{Edge}'$ is the finite subset of $\mathbf{Edge}$ that consists of all the edges that are potentially part of the program. The fact that the algorithm always computes the most precise result heavily depends on the existence of the unique least fixed point μ of F. It is easy to show that the generic algorithm computes this least fixed point μ. As the least fixed point is the best possible result with respect to the precision of the abstract domain, it is always the most precise regardless of the non-deterministic choices made in line 13.

Proof For a given IL program with an associated set of addresses $\mathbf{L}$ the set

$$\begin{aligned}
&\mathbf{Edge}^{\bigstar} = \\
&= \{(\ell, stmt, \ell') \mid \ell, \ell' \in \mathbf{L} \text{ with } [stmt]^{\ell}_{\ell'}, \text{ where } stmt \text{ is } v := e \text{ or } m[e_1] := e_2\} \\
&\cup \{(\ell, \mathsf{assume}\ (e_1 = 0_1), \ell'), (\ell, \mathsf{assume}\ (e_1 \neq 0_1 \wedge e_2 = \ell''), \ell'') \mid \\
&\qquad\qquad\qquad\qquad \ell, \ell' \in \mathbf{L} \text{ with } [\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]^{\ell}_{\ell'}, \ell'' \in \mathbf{L}\}
\end{aligned}$$

is a finite subset of $\mathbf{Edge}$. Therefore $\mathbf{2}^{\mathbf{Edge}^{\bigstar}}$ satisfies the ACC. The lattice $\mathbf{L} \to A$ satisfies the ACC because A satisfies the ACC and $\mathbf{L}$ is finite. It follows that $(\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}^{\bigstar}}$ satisfies the ACC.

The function $H : \Big((\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}^\star}\Big) \to \Big((\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}^\star}\Big)$ with

$$H(D, G) \mapsto (D', G')$$

denotes one iteration of the `while` loop of the generic algorithm, where (D, G) is updated to (D', G') according to the body of the loop. It is apparent that H is extensive, i.e., $\forall (D, G) \in (\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}^\star}.\, (D, G) \sqsubseteq H(D, G)$.

We define $\bot_H$ to be the value of (D, G) after the initialization, i.e., after executing lines 2 to 5, and $H^n(\bot_H)$ to be the value of (D, G) after the nth iteration of the while loop. We extend the definition of H such that $H^{n+1}(\bot_H) = H^n(\bot_H)$ if the loop terminated after the nth iteration of the while loop. Clearly, the sequence $(H^n(\bot_H))_{n \in \mathbb{N}_0}$ is an ascending chain. It eventually stabilizes, i.e., there is an n_0 such that $\forall n \geq n_0.\, H^n(\bot_H) = H^{n_0}(\bot_H)$, as $(\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}^\star}$ satisfies the ACC. The termination of the generic algorithm immediately follows.

Initial values ι^a of F are defined as $\iota^{\mathbf{start}}$ for $a = \mathbf{start}$ and $\bot$ for $a \neq \mathbf{start}$. For the given program we can assume that the type of the function F is $\Big((\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}^\star}\Big) \to \Big((\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}^\star}\Big)$. We denote by $\mu_{\iota^{\mathbf{start}}}$ the least fixed point of F for the initial value $\iota^{\mathbf{start}}$. We will now show that the generic algorithm computes $\mu_{\iota^{\mathbf{start}}}$.

It can easily be seen that

$$\forall (D, G) \in \Big((\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}^\star}\Big) \to \Big((\mathbf{L} \to A) \times \mathbf{2}^{\mathbf{Edge}^\star}\Big).$$
$$H(D, G) \sqsubseteq F(D, G) \quad (3.7)$$

From $\bot_H \sqsubseteq \mu_{\iota^{\mathbf{start}}}$ and from (3.7) we have $\forall n \in \mathbb{N}_0.\, H^n(\bot_H) \sqsubseteq \mu_{\iota^{\mathbf{start}}}$ by induction on n. Now it remains to show that $H^{n_0}(\bot_H)$ is $\mu_{\iota^{\mathbf{start}}}$. We prove this by contradiction. We assume that $H^{n_0}(\bot_H) \sqsubset \mu_{\iota^{\mathbf{start}}}$. It follows that $H^{n_0}(\bot_H) \neq F(H^{n_0}(\bot_H))$. From (3.7) we have

$$H^{n_0}(\bot_H) = H^{n_0+1}(\bot_H) = H(H^{n_0}(\bot_H)) \sqsubseteq F(H^{n_0}(\bot_H)).$$

Therefore $H^{n_0}(\bot_H) \sqsubset F(H^{n_0}(\bot_H))$. This means that there is an address $\ell \in \mathbf{L}$, such that $G \not\supseteq \mathbf{resolve}_\ell(D(\ell))$, or an edge $(\ell', stmt, \ell) \in G$, such that $D(\ell) \not\sqsupseteq \widehat{\mathbf{post}}[\![stmt]\!]\ (D(\ell')) \sqcup \iota^\ell$. This is a contradiction to the assumption that the algorithm terminated at least after the n_0th iteration of the loop.

□

3.5.2 Worklist Algorithm

The worklist algorithm shown in Figure 3.3 is a specific strategy for executing the generic algorithm, where the set of edges $G \subseteq \mathbf{Edge}$ is not kept as a variable, but implicit in the abstract values of the program locations. The initialization of D (lines 3, 4) is the same as in the generic algorithm. The algorithm maintains a worklist W, where it stores the edges for which data flow facts should be propagated later on. Every time the algorithm updates the information $D(\ell)$ at a location a (lines 4, 9), it calls the **resolve** operator (lines 5, 10) to calculate the edges which should be added to W. In every iteration of the main loop (lines 6 to 10) the algorithm non-deterministically picks an edge from the worklist by calling choose and then shortens the worklist by calling rest (line 7). Subsequently, it checks for the received edge $(\ell', stmt, \ell)$, if an update is necessary (line 8), and in the case it is, it proceeds as already described.

From the correctness of the generic algorithm (3.2) we obtain the correctness of the worklist algorithm:

Corollary 3.8 *Given an IL program, where* $\{s \in \mathbf{State} \mid s(pc) = \mathbf{start}\} \subseteq \gamma(\iota^{\mathbf{start}})$, *the worklist control flow reconstruction algorithm (as defined in Figure 3.3) computes a sound overapproximation of the CFA and terminates in finite time. Furthermore it returns the most precise result with respect to the precision of the abstract domain A regardless of the non-deterministic choices made in line 7.*

The worklist terminates because A satisfies the ascending chain condition. As the generic algorithm can always simulate the updates made by the worklist algorithm, the result computed by the worklist algorithm is always less or equal

```
Input: an IL program, its set of addresses $\mathbf{L}$ including $\mathbf{start}$, and the
       abstract domain $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$ together with an
       initial value $\iota^{\mathbf{start}}$
Output: a control flow automaton
begin
    forall the $\ell \in \mathbf{L} \setminus \{\mathbf{start}\}$ do
        $D(\ell) := \bot$;
    $D(\mathbf{start}) := \iota^{\mathbf{start}}$;
    $W := \mathbf{resolve}_{\mathbf{start}}(D(\mathbf{start}))$;
    while $W \neq \emptyset$ do
        $((\ell', stmt, \ell), W) := (\texttt{choose}(W), \texttt{rest}(W))$;
        if $\widehat{\mathbf{post}}[\![stmt]\!](D(\ell')) \not\sqsubseteq D(\ell)$ then
            $D(\ell) := \widehat{\mathbf{post}}[\![stmt]\!](D(\ell')) \sqcup D(\ell)$;
            $W := \texttt{add}(W, \mathbf{resolve}_{\ell}(D(\ell)))$;
    $G := \emptyset$;
    forall the $\ell \in \mathbf{L}$ do
        $G := G \cup \mathbf{resolve}_{\ell}(D(\ell))$;
    return $G$;
end
```

Figure 3.3: Worklist Control Flow Reconstruction Algorithm.

to the result of the generic algorithm, which is the least fixed point of F. On the other hand it can be shown that if the algorithm terminates, the result is greater or equal to the least fixed point of F.

Proof We denote by the function

$$W : (\mathbf{L} \to A) \to (\mathbf{L} \to A)$$

with

$$W(D) \mapsto (D')$$

one iteration of the `while` loop of the worklist algorithm, where D is updated to D' according to the body of the loop. It is obvious that W is extensive, i.e., $\forall D \in \mathbf{L} \to A.\, D \sqsubseteq W(D)$. Analogously to the proof for the generic algorithm, we define $\bot_H$ to be the value of D after the initialization, i.e., after executing lines 3 and 4. We define $W^n(\bot_H)$ to be the value of D after the nth iteration of the while loop, and we define $W^{n+1}(\bot_H)$ to be $W^n(\bot_H)$, if the loop terminated after the nth iteration of the while loop. Clearly the sequence $(W^n(\bot_H))_{\mathbb{N}_0}$ is an ascending chain. It eventually stabilizes, i.e., there is an n_0 such that $\forall n \geq n_0.\, W^n(\bot_H) = W^{n_0}(\bot_H)$, as $\mathbf{L} \to A$ satisfies the ACC (see the proof of Theorem 3.7). The termination of the worklist algorithm immediately follows.

We denote the result of the worklist algorithm $W^{n_0}(\bot_H)$ by $D_\bigstar$. We have

$$\left(D_\bigstar, \bigcup_{\ell \in \mathbf{L}} \mathbf{resolve}_\ell(D_\bigstar(\ell))\right) \sqsubseteq \mu_{\iota^{\mathsf{start}}},$$

as the lines 9 and 10 of the worklist algorithm can be simulated by the generic algorithm by choosing the respective necessary updates (see proof of Theorem 3.7 for $\mu_{\iota^{\mathsf{start}}}$).

It remains to show that $\left(D_\bigstar, \bigcup_{\ell \in \mathbf{L}} \mathbf{resolve}_\ell(D_\bigstar(\ell))\right) \sqsupseteq \mu_{\iota^{\mathsf{start}}}$. We will show that

$$D_\bigstar(\mathbf{start}) \sqsupseteq \iota^{\mathbf{start}} \tag{3.8}$$

and

$$\forall \ell' \in \mathbf{L}.\forall(\ell', stmt, \ell) \in \mathbf{resolve}_{\ell'}(D_\bigstar(\ell')). \quad D_\bigstar(\ell) \sqsupseteq \widehat{\mathbf{post}}[\![stmt]\!](D_\bigstar(\ell')). \tag{3.9}$$

Having established (3.8) and (3.9), we know that $\left(D_\bigstar, \bigcup_{\ell \in \mathbf{L}} \mathbf{resolve}_\ell(D_\bigstar(\ell))\right)$ is a solution to the constraint system (3.1) and (3.2) and therefore greater than the least fixed point $\mu_{\iota^{\mathbf{start}}}$. We have (3.8) by the fact that $\bot_H(\mathbf{start}) = \iota^{\mathbf{start}}$, and by the fact that W is extensive, as $W^{n_0}(\bot_H) = D_\bigstar$. Now we prove (3.9) by contradiction. We assume that there are $\ell', \ell \in \mathbf{L}$, such that

$$(\ell', stmt, \ell) \in \mathbf{resolve}_{\ell'}(D_\bigstar(\ell')),$$

and

$$D_\bigstar(\ell) \not\sqsupseteq \widehat{\mathbf{post}}[\![stmt]\!](D_\bigstar(\ell')).$$

We proceed by case distinction on the last time that $D_\bigstar(\ell')$ was updated:

- line 3. There is no edge $(\ell', stmt, \ell) \in \mathbf{resolve}_{a'}(D_\bigstar(\ell'))$ as $D_\bigstar(\ell') = \bot_H$ holds throughout the execution of the algorithm, and therefore **resolve** never adds an edge. This is a contradiction.
- line 4. ℓ' is **start**. $D_\bigstar(\ell) \sqsupseteq \widehat{\mathbf{post}}[\![stmt]\!](D(\mathbf{start}))$ is ensured in a later step as the edge $(\mathbf{start}, stmt, \ell) \in \mathbf{resolve}_{\mathbf{start}}(D_\bigstar(\mathbf{start}))$ is added to the worklist (line 5), and remains invariant after that, as $D_\bigstar(\mathbf{start})$ does not change until termination. This leads to a contradiction.
- line 9. $D_\bigstar(\ell) \sqsupseteq \widehat{\mathbf{post}}[\![stmt]\!](D(\ell'))$ is ensured in a later step as the control flow edge $(\ell', stmt, \ell) \in \mathbf{resolve}_{\ell'}(D_\bigstar(\ell'))$ is added to the worklist (line 10), and remains invariant after that as $D_\bigstar(\ell')$ does not change until termination. This leads to a contradiction. □

Note that if the abstract domain A does not satisfy the ascending chain condition, it is possible to enhance the algorithms by using a widening operator to guarantee termination of the analysis. Such an algorithm would achieve a valid overapproximation of the CFA but lose the best approximation result stated in the above theorems, due to the imprecision induced by widening.

3.6 Related Work

The literature contains a number of practical approaches to disassembly and control flow reconstruction, which do not try to formulate a generalizable strategy. Schwarz et al. [123] describe a technique that uses an improved linear sweep disassembly algorithm, using relocation information to avoid misinterpreting data in a code segment. Subsequently, they run a recursive traversal algorithm on each function and compare results, but no attempt is made to recover from mismatching disassembly results. Harris and Miller [66] rely on identifying compiler idioms to detect procedures in the binary and to resolve indirect jumps introduced by jump tables. Cifuentes and van Emmerik [35] present a method to analyze jump tables by backward slicing through register assignments and computing compound target expressions for the indirect jumps. These compound expressions are then matched against three compiler-specific patterns of implementing switch statements. In the framework proposed in this chapter, jump tables do not need special treatment. Indirect branches through jump tables can be resolved using an interval analysis as data flow domain, for example (see Section 5.3.4).

There have also been several proposals for more general frameworks for reconstructing the control flow from binaries [50, 78, 133]. For integrating data flow analysis with disassembly, De Sutter et al. [50] suggested to initially connect all indirect jumps to a virtual *unknown* node for indirect jumps, which effectively overapproximates the control flow graph. In an iterative process, they use constant propagation on the overapproximated graph to show infeasibility of most overapproximated edges, which can then be removed. This approach is inspired by the solution of Chang et al. [27] to the similar problem of treating unknown

```
1:  $x := 5$
2:  if  jmp $x$
3:  $x := x - 2$
4:  if  jmp 2
5:  halt
```

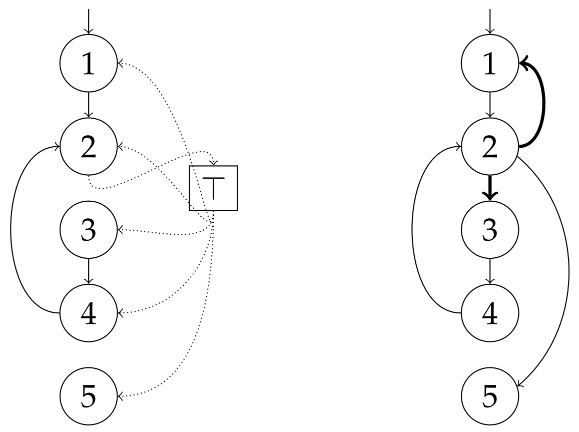


Figure 3.4: Adding an unknown node (⊤) with unlabeled edges leads to additional possible values for x at the indirect jump.

external library functions in the analysis of C programs. Figure 3.4 exemplifies De Sutter et al.'s method by applying it to a snippet of pseudo-assembly code. The middle of the figure depicts the corresponding initial control flow graph, where the indirect jump at line 2 is connected to the unknown node (⊤). There are outgoing edges from the unknown node to all statements, since every address is a potential jump target in the general case of stripped code without relocation information. Calculating the possible values of x, it can be seen that x can in fact take the concrete values $5, 3, 1, -1, \ldots$ at the entry of line 2 in the overapproximated program. Thus a program analysis operating on this initial overapproximation can only conclude that addresses 2 and 4 are no targets of the jump, but cannot remove the overapproximated edges to addresses 1 and 3. The final CFG reconstructed by this method, shown on the right of Figure 3.4, consequently contains the infeasible edges (2,1) and (2,3) (drawn in bold).

In other related work, Theiling [133] proposes a bottom-up disassembly strategy, which assumes architectures where all jump targets can be computed directly form the instruction, effectively disallowing indirect jumps. For extending his method to indirect jumps, he suggests the use of an overapproximating unknown node in the manner of Chang et al. [27] and De Sutter et al.[50].

Kästner and Wilhelm [78] describe a top-down strategy for structuring executables into procedures and basic blocks. For this to work, they require that code areas of procedures must not overlap, that there must be no data between or inside procedures, and that explicit labels for all possible targets of indirect jumps are present. Compilers, however, commonly generate procedures with overlapping entry and exit points, even if the control flow graphs of the procedures are completely separate, so their top-down structuring approach cannot be used in general without specific assumptions about the compiler or target architecture.

Vigna and Kruegel et al. [85, 135] attack the problem of disassembling obfuscated executables, as a direct response to the anti-disassembly techniques proposed by Linn and Debray [90]. Their approach is a refinement of the recursive traversal approach as used in disassemblers such as IDA Pro: After using heuristics to detect function entry points, they start recursive traversal on all addresses in a function to obtain a first, fragmented control flow graph containing a large amount of invalid instructions. After this phase, they consolidate the CFG by removing overlapping instructions and favoring instructions connected by branch instructions. Statistical techniques for identifying likely code are then used to fill up "gaps" of unreached instructions. The approach is not targeted at extracting a connected control flow graph for a sound static analysis, as the heuristic methods are likely to generate a disconnected control flow graph. Limitations are that procedures cannot be intertwined and that in the final CFG, instructions cannot overlap; both issues are challenges identified in Section 1.2 and are handled by the framework proposed in this dissertation.

The most well known and successful approach to static analysis of executables to date is the CodeSurfer/x86 project [6, 7, 117]. For disassembly, they rely on the capabilities of the commercial disassembler IDA Pro. Generally, they assume a standard compilation model for binaries, which guarantees correct disassembly by IDA Pro. They acknowledge that IDA Pro's output can be incomplete and do connect missing edges from indirect calls, yet they lack a complete loop to disassemble previously unprocessed branch targets. Furthermore, IDA Pro is prone not only to omitting control flow edges but also to producing false positives of code that is never executed. Thus the soundness of CodeSurfer/x86 is severely impacted by errors introduced by the heuristics based disassembly strategy of IDA Pro.

Although operating at higher language levels, the decompilation approach of Chang et al. [26] is similar in spirit to the framework presented in this chapter. They connect abstract interpreters operating at different language levels, executing them simultaneously. One can interpret the combined data flow analysis and control flow reconstruction as separate decompilation stages of their framework. The framework presented in this chapter does not restrict the execution order, however, but allows nondeterministic fixed point iteration over both analyses while still maintaining that the resulting CFA is optimal.

Finally, Nanda et al. [105] follow a different route and use a mixture of static analysis and dynamic binary instrumentation to disassemble Windows executables. They use a recursive traversal algorithm to statically disassemble as much code as possible, and instrument each indirect branch, including the call sites for user callbacks in system libraries. The instrumentation code then dynamically calculates the branch target and invokes the disassembly algorithm on the target if it has not yet been disassembled. They do not support overlapping instructions and always include both branches of conditional jumps in the disassembly regardless of whether or not the branches are feasible at runtime, although these limitations appear to be merely design decisions for maintaining a low runtime overhead. The approach is targeted at supplying dynamic instrumentation frameworks with additional information. Since only a single execution is monitored, it does not necessarily disassemble the entire program. The authors report a coverage of disassembled instructions of about 69% to 96%.

Chapter 4

Bounded Address Tracking

This chapter introduces *Bounded Address Tracking*, an abstract domain specifically targeted at unstructured low level code lacking types. It is designed to serve as data flow domain in the control flow reconstruction framework presented in Chapter 3, and it is the principal abstract domain used for control flow reconstruction and verification in the device driver experiments of Section 6.1. It's high precision yields both high quality control flow automata and little false positives for static assertion verification on moderately sized programs.

Bounded Address Tracking makes no distinction between pointers and integers, as both are indistinguishable in low level code. It uses a memory model based on separate memory regions, and represents values of registers and memory locations as consisting of a region identifier and an offset. This allows to support address arithmetic, which is pervasive in binaries. As low level code does not have a reliable concept of procedures, Bounded Address Tracking performs a path sensitive analysis to achieve context sensitivity in procedures, if they exist. The precise modeling of memory locations even allows to handle the modification of return addresses on the stack. Path sensitivity further avoids the creation of summary regions in the heap and subsequent weak updates, which leads to a very precise analysis of heap structures. To limit path explosion and ensure practical feasibility, Bounded Address Tracking uses an adjustable bound on the number of values that are tracked per variable per location.

4.1 Precision Requirements

The translation of guarded jumps to labeled edges requires a precise evaluation of the target expression. Otherwise, the **resolve** operator (see Section 3.4.1) will introduce a spurious control flow edge for every address that is not a concrete jump target but still contained in the overapproximation of the evaluated target expression. Spurious control flow edges are likely to point into code or data sections never meant to be executed, which initiates a cascading loss of precision. If **resolve** has no information at all about the target values of a guarded jump, a sound overapproximation of the control flow has to include edges from the jump to *all* program locations in the entire memory, even those outside the set of valid program locations. In the implicitly extended program mapping that introduced assert 0_1 statements in illegal code locations (see Section 2.5), each of the edges would lead to a spurious assertion failure.

To avoid overapproximation and spurious control flow, jump target expressions need to be precisely evaluated and the result has to be represented with its concrete value. Only a concrete value can uniquely identify an address in the program. Even interval representations can be too coarse; for instance, if an indirect branch can jump to the addresses 1000 and 1007 at runtime, a static interval analysis can only approximate the target address to be in the interval $[1000; 1007]$. Using this approximation to build the control flow graph would introduce six spurious edges that add additional imprecision to the analysis. The use of strided intervals as in VSA [7] does not provide a general solution to this problem. In the example, both addresses can be precisely represented as the interval $[1000; 1007]$ with a stride of 8. In presence of a third value that is not aligned with the stride, such as 1011 in the example, the strided interval widens to $[1000; 1011]$ with stride 4. This includes the overapproximated address 1003 that can again contain invalid code. Since x86 code is not aligned, it is very unlikely that a strided interval is able to precisely represent a set of branch targets without including additional values.

Apart from the precision requirements for control flow reconstruction, pointer reasoning in machine code also requires an exact representation of address values. Address arithmetic is pervasive, so offsets within the stack or the heap have to be precisely tracked. Furthermore, the lack of types prohibits a limited overapproximation of points-to sets for unknown values. In regular source based static analysis, an unknown pointer may only point to all variables of a matching type. In untyped assembly code, every value can be dereferenced and an unknown pointer may point to any location in the entire memory, including code. A write access to an unknown pointer could thus write to any address in the entire memory. Overapproximating the effects of such an update again causes a cascading loss of precision. Since return values and function pointers are stored in memory, this information loss in memory also affects control flow information. Therefore, it is apparent that an analysis has to precisely represent address values if it is to reason about the semantics of machine code.

A third issue in executables is that procedures are not necessarily a reliable concept (see Section 1.2). Therefore, the classical call strings approach is unsound on executables. In procedures that are called from different call sites, it is important to keep calling contexts separate, however. Since return instructions are just translated to indirect jumps, the return address is just another address value stored on the stack. If contexts are not kept separate, these addresses would be merged and the information that procedures (usually) return to the location from which they were called, would be lost. Path sensitivity subsumes context sensitivity, but does not require assumptions about procedure layout.

Bounded Address Tracking was designed with these challenges in mind. It represents the values in registers and memory locations with a symbolic base address and a concrete offset, thus supporting the kind of pointer arithmetic found in binaries. Path sensitivity allows it to separate calling contexts of procedures and to uniquely identify allocated heap memory. Scalability is ensured using bounds on the number of variable values. Context sensitivity in the analysis of procedures is therefore maintained as long as the number of call sites lies below the configurable value bound.

4.2 Partitioned Memory Model

The virtual memory available to a process is organized as one large, continuous array. The stack, the heap, and global variables all share this address space, which is reflected in the rules AssignMem and Alloc of the concrete IL semantics (Table 2.1). On the other hand, there is usually an implicit partitioning of the memory into logical regions.

Memory Regions. When a process is loaded, the runtime environment (the OS and standard libraries) sets up the stack and the heap in the virtual memory. The initial absolute positions of the stack and heap base are not guaranteed to be of a particular value, but the system sets them up in such a way that they do not interfere, and it installs buffer pages between them to detect overflows. Furthermore, correct implementations of malloc and similar API functions guarantee that allocated memory blocks in the heap do not overlap, but again give no guarantees about the absolute address of allocated blocks. A static analysis reasoning about stack and heap faces the problem that it cannot fix the base addresses to a certain absolute value but has to represent all possible base addresses symbolically.

It is reasonable to assume that stack and allocated heap regions are usually disjoint, however, due to the OS and standard library implementations. This assumption is the basis of the partitioned memory model, which abstracts the flat model of a single address space part of the concrete semantics in Chapter 2. The partitioned memory model is based on a set $\mathbf{R}$ of separate *memory regions*:

- The global region, containing code, global variables, and static data,
- a single stack, holding local variables, parameters, and return addresses at runtime,
- and an infinite number of heap regions, which correspond to memory blocks allocated using malloc.

Thus, every memory address is abstracted to a pair from $\mathbf{R} \times \mathbb{I}_*$, consisting of a memory region identifier, which serves as a symbolic base address, and a bit-vector offset. Pointers into the global region are denoted by $(\mathsf{global}, \mathit{offset})$; the stack pointer is assumed to be initialized to a value of $(\mathsf{stack}, 0)$. Subsequent modifications to the stack pointer then change the offset, but let it stay within the stack region. In x86, the stack grows downward, so the stack pointer will always have negative offsets within valid code. The number of heap regions is unbounded, and a fresh heap region is created by any call to malloc. A fresh identifier tags the individual heap region, creating pointers such as $(\mathsf{alloc}_{id}, \mathit{offset})$.

Soundness. This memory model presents an abstraction of the flat memory model, since it treats the initial base addresses of the stack and allocated heap regions symbolically. The relative positions of regions to each other are ignored. If for whatever reason the partitioned memory model is too imprecise for the kind of code being analyzed, it can emulate the flat memory model by initializing the stack pointer and any newly allocated memory to addresses in the global memory region.

For a fully sound abstraction of the flat memory model, semantics that use the partitioned memory model have to check bounds on each memory access, i.e., compare the offset with the size of the allocated heap region, the maximum stack size, or the address range available to global variables and static data. If the bound is exceeded, all other memory regions may be written to, since the relative position of regions is unknown. Note that the abstract semantics of Bounded Address Tracking, which will be defined in Section 4.4, does not perform this bounds check – therefore it is sound only under the assumption that no pointers escape their allocated memory bounds.

Integer Values. In machine code, there is no difference between integers and pointers, all variables that are of the architecture-defined address bit length can be interpreted as both. This ambiguity can be represented within the partitioned memory model by interpreting the global address space as integer values. Global addresses start at virtual address 0 and are not computed relative to some allocated base address, thus they are indistinguishable from integer values. This interpretation follows the same idea as the memory model of VSA [7]. A particular difference between the memory models is that the partitioned memory model, unlike VSA, does not make the assumption of isolated procedure stack frames, but uses a single region for the entire stack instead.

Note that for ease of explanation the following assumes an architecture with an address width of 32 bits; by substituting this value, the domain of Bounded Address Tracking can be adapted to different architectures, such as 16 or 64 bit.

4.3 Abstract Domain of Address Valuations

Bounded Address Tracking maintains states as partial variable valuations, i.e., abstract states store abstract addresses for a subset of the registers and memory locations used by the program.

Abstract Addresses. The domain of abstract values abstracts from the partitioned model of memory addresses of the form $(\mathit{region}, \mathit{offset})$. It provides a mechanism for abstracting multiple addresses with a single, less precise abstract value for representing coarse information or the complete lack of it. Integers of bit lengths other than the address width can be added to the same domain as well, to provide a unified view of the values dealt with in the analysis. To this end, the set of addresses in the partitioned memory model is abstracted to a complete lattice of abstract addresses and integer values. The lattice includes a top address element $(\top_R, \top_{32})$ representing a memory address with the unknown region $\top_R$ and unknown offset $\top_{32}$. Furthermore, the abstract memory model provides an intermediate level of pointers with known region but unknown offset of the form $(\mathit{region}, \top_{32})$, which represents the join of different addresses within the same region (e.g., $(r, 4_{32}) \sqcup (r, 8_{32}) = (r, \top_{32})$). Integers of a bit length other than the address bit length use the global region only, as they cannot point to valid memory locations. There is still an abstract value with an unknown region such as $(\top_R, \top_8)$ for every other bit length, however, which is

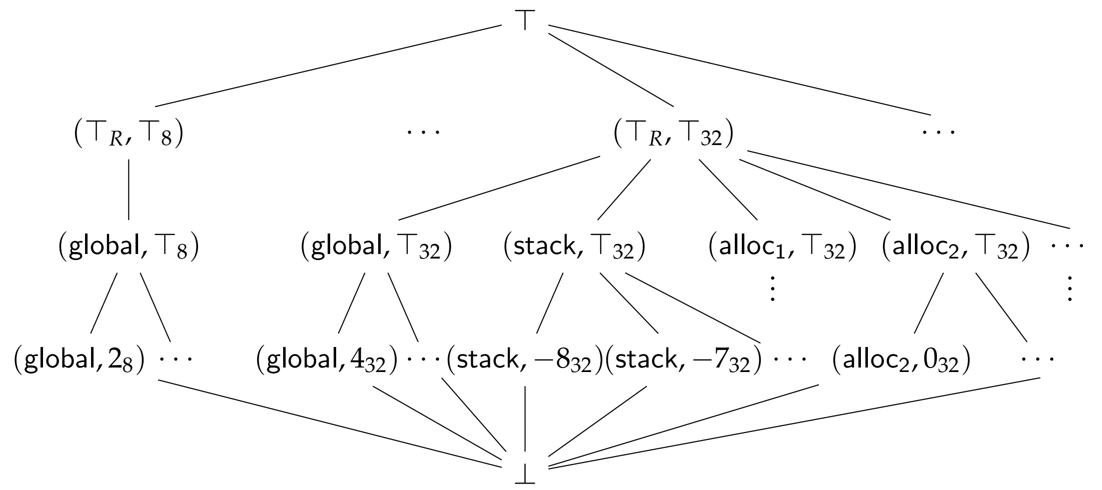


Figure 4.1: Diagram of the lattice of abstract addresses and values $\hat{A}$.

used for representing fragments or extensions of address width pointers. This is necessary for sound overapproximation of splitting and recombination of pointers, which would otherwise incorrectly yield an unknown pointer into the global region (see Section 4.6.1).

For a maximum supported bit length of m, the set of abstract memory addresses $\hat{A}$ is thus defined as

$$\hat{A} = \{\top\} \cup (\{\top_R, \mathsf{global}\} \times \top_{1,\ldots,m}) \cup (\{\mathsf{global}\} \times \mathbb{I}_{1,\ldots,m}) \cup (\mathbf{R} \times (\mathbb{I}_{32} \cup \top_{32})),$$

where $\mathbf{R}$ denotes the set of memory regions and $\top_{1,\ldots,m}$ the set of top elements for all supported bit lengths. The resulting infinite lattice for $\hat{A}$ is sketched in Figure 4.1. Note that the global region contains infinitely many abstract addresses, due to infinitely many possible bit lengths; moreover, there are infinitely many addresses with the architecture dependent address bit length, as there is an unbounded number of heap regions.

Abstract States. The analysis overapproximates the set of reachable concrete states of the program by calculating a fixpoint over the abstract states. For each reachable program location, it computes multiple abstract states that approximate the different path contexts for this location. Using $\hat{A}$ as value domain for registers and memory locations, the abstract states are the Cartesian product of the individual valuations. The set of abstract states is defined as

$$\hat{S} = \mathbf{Loc} \times \widehat{\mathbf{Val}} \times \widehat{\mathbf{Store}},$$

consisting of the current program counter value, an abstract register valuation $\widehat{\mathbf{Val}} := V \to \hat{A}$ and an abstract store $\widehat{\mathbf{Store}} := \hat{A} \to \hat{A}$.

Initial State. The initial state at the entry point of the executable is initialized to $(pc \to \mathbf{start}, \{esp \to (\mathsf{stack}, 0_{32})\}, \{(\mathsf{stack}, 0_{32}) \to \mathbf{end}, (\mathsf{global}, \ell_0) \to d_0), \ldots, (\mathsf{global}, \ell_n) \to d_n)\}), \ell_0, \ldots, \ell_n \in \mathbf{L}, d_0, \ldots, d_n \in \mathbb{I}_8 :$

- The program counter is initialized to the IL program's starting location **start**.
- The register valuation is initialized to hold a valid initial abstract value for the stack pointer *esp*. The stack grows downward in x86, thus subsequent stack accesses will operate on negative indices in the stack region.
- The stack in the abstract store is initialized to hold the single value **end**, which points to a halt statement for catching control flow after the program's main procedure returns. Initially *esp* points to this value, thus the IL statements corresponding to the top-level return instruction read this address from the stack and jumps to it. Usually programs return control to the operating system when terminating, so this artificial halt statement provides a way to actually end the execution path.
- The global memory region is initialized to hold the static data present in the executable, e.g., initial values for global variables, integer or string constants. In the description of the initial state, $\ell_0, \ldots, \ell_n$ denote static data locations in the executable and $d_0, \ldots, d_n$ their respective values.

- All registers and memory locations not shown in the initial state are implicitly set to $(\top_R, \top_{32})$. This includes all offsets in all newly allocated heap regions.

The initial state provides a minimal execution environment for any program by setting up a valid stack, and installing a halt statement for catching returning control flow from the main function. A practical implementation of an analysis environment that sets up the initial state through a prologue of IL statements, instead of explicitly fixing it, is discussed in Section 5.1.4.

An efficient implementation of abstract states cannot explicitly store all static data of the executable in the abstract state. Instead, the static data is initialized lazily. On a write access to a static data location (necessarily an address in the global memory region), the abstract state is updated with the new value for that memory location. On a read access that is determined to point into the part of the global region that is occupied by static data, it is first checked whether a value for that location is explicitly present in the abstract state. If so, this value is returned, otherwise the initial static value is read from the executable directly.

4.4 Abstract Semantics

The abstract semantics of Bounded Address Tracking is given using the bounding operator **bound** $:: \hat{S} \to (V \cup \hat{A}) \to \hat{S}$ (defined in Table 4.1), the abstract evaluation operator $\widehat{\mathbf{eval}} :: \mathbf{Exp} \to \hat{S} \to \hat{A}$ (defined in Tables 4.2 and 4.3), and the abstract transfer function $\widehat{\mathbf{post}} :: \mathbf{Stmt} \to \hat{S} \to 2^{\hat{S}}$ from statements and abstract states to sets of abstract states (defined in Table 4.4). A worklist algorithm extended to apply and adapt precision information [18] (here, bounds over the number of abstract values) enforces the bound for all registers and memory locations before calculating the abstract transfer function. The formalization of the analysis within the Jakstab framework, which uses the worklist algorithm from [18], is discussed in Section 5.3.2.

For notational convenience, this chapter uses a slightly different notation, but Bounded Address Tracking can instantiate the control flow reconstruction framework given in Chapter 3 as the abstract data flow domain $(\hat{S}, \bot_{\hat{S}}, \top_{\hat{S}}, \sqcap_{\hat{S}}, \sqcup_{\hat{S}}, \sqsubseteq_{\hat{S}}, \widetilde{\mathbf{post}}, \widetilde{\mathbf{eval}}, \gamma)$. For this purpose, the lattice and its operations are defined as the product of the lattices for all variable mappings. The concretization function γ maps abstract states to sets of concrete states by enumerating all combinations of possible base addresses for abstract memory regions and all possible concrete values for unknown offsets or regions. The abstract post operator $\widetilde{\mathbf{post}}$ is a composition of $\widehat{\mathbf{post}}$ and **bound** applied to all variables. The abstract evaluation operator $\widetilde{\mathbf{eval}}$ follows the definition of Chapter 3 and evaluates expressions to sets of integers by concretizing the result of $\widehat{\mathbf{eval}}$. It is defined as

$$\widetilde{\mathbf{eval}}[\![e]\!](a) := \gamma_{\hat{A}}(\widehat{\mathbf{eval}}[\![e]\!](a)),$$

where $\gamma_{\hat{A}} :: \hat{A} \to \mathbb{Z}$ concretizes a single abstract address:

$$\gamma_{\hat{A}}\big((r, o)\big) := \begin{cases} o & \text{if } r = \mathsf{global} \\ \mathbb{Z} & \text{otherwise} \end{cases}$$

4.4.1 Bounded Path Sensitivity

Bounded Address Tracking is path sensitive, i.e., it does not join abstract states when control flow combines from different paths, e.g., after a conditional block, a loop, or at the beginning of a procedure. This way, calling contexts are kept separate, the relations between variable values are maintained, and the abstract states remain very precise. The very fine grained separation of contexts by paths even allows to precisely model and handle modifications of return addresses on the stack. Such behavior is not captured by classical call string approaches.

Due to the path explosion problem, path sensitivity in general is prohibitively expensive, and it is necessary to take some precautions to make the analysis feasible. To ensure termination and speed up the computation of a fixpoint, the

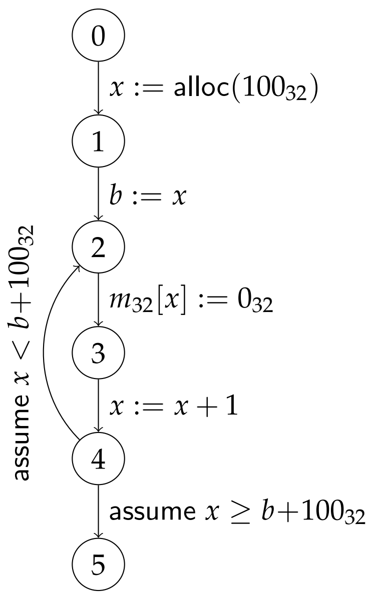
Bound

$$\mathbf{bound}(s, x_b) := \mathsf{let}(r, o) = s(x)$$
$$\begin{cases} s & \text{if } \|\{s(x) \mid s \in \{s' | s'(pc) = \ell\}\}\| \leq k \\ s[x \mapsto (\top_R, \top_b)] & \text{if } \|\{r' \mid \exists s' \in \{s'' | s''(pc) = \ell\}.(r', o') = s'(x)\}\| > k \\ s[x \mapsto (r, \top_b)] & \text{otherwise} \end{cases}$$

Table 4.1: Definition of the bound operator.

analysis uses bounds on the number of values tracked per variable (hence the name *Bounded Address Tracking*).

In particular, the analysis bounds the number of abstract values *per variable per location* that it explicitly tracks, and, if the bound is exceeded, it performs widening in two steps. Before calculating abstract successors for a state s at location ℓ, the analysis checks for each register or memory location x whether the total number of unique abstract values for x in all reached states at ℓ exceeds the configured bound k. The number of values for x is thus collected over all paths that pass through ℓ. If the number is below the bound, the variable remains unchanged (see the first case of of the Bound rule in Table 4.1). If the bound is exceeded, the value of x in state s is widened to $(r, \top_b)$, where b is the bit length of x and r is the memory region of x's value in s (third case). This generalized value represents all values within the memory region r. If this generalization does not include all possible values for x at ℓ, more values will keep accumulating which have a memory region other than r. If at some point also the number of unique memory regions exceeds the bound k, then x is widened to $(\top_R, \top_{32})$ in s (second case of Bound). This value overapproximates all possible values for x, thus no additional values for x can accumulate, and the analysis eventually terminates.


$x := \mathsf{alloc}(100_{32})$
$b := x$
$m_{32}[x] := 0_{32}$
$x := x + 1$
assume $x \geq b{+}100_{32}$
assume $x < b{+}100_{32}$

ℓ	# x	# b	x
0	1	1	$(\top_R, \top_{32})$
1	1	1	$(\mathsf{alloc}_1, 0_{32})$
2	6	1	$(\mathsf{alloc}_1, 0_{32})$
3	6	1	$(\mathsf{alloc}_1, \top_{32}), \ldots, (\mathsf{alloc}_0, \top_{32})$
4	6	1	$(\mathsf{alloc}_1, \top_{32}), \ldots, (\mathsf{alloc}_0, \top_{32})$
5	1	1	$(\mathsf{alloc}_1, \top_{32})$

x, # b: Number of unique values for x and b.
x: Abstract addresses for x in states at ℓ.

Figure 4.2: Example code fragment and final state space.

Consider the example code on the left of Figure 4.2, represented as a control flow automaton, which uses the variables x and b and allocates and writes to a heap region. The single initial abstract state is $(0, \{x \to (\top_R, \top_{32}), b \to (\top_R, \top_{32})\}, \emptyset)$, so there is one unique value per variable. Assume the bound k is set to 5. After creating a new abstract heap region and copying the pointer into b, the analysis enumerates states in the loop at locations 2, 3, 4 while the edge $(4, \mathsf{assume}\ x \geq b + 100_{32}, 5)$ remains infeasible. When the state $(2, \{x \to (\mathsf{global}, 5_{32}), b \to (\mathsf{global}, 0_{32})\}, \{(\mathsf{alloc}_1, 0_{32}) \to (\mathsf{global}, 0_{32}), \ldots\})$ is reached, the analysis counts 6 unique values for x in location 2, and widens x to $(\mathsf{alloc}_1, \top_{32})$. This causes a weak update to alloc_1 once x is dereferenced. At the end of the loop, both assume edges are now feasible, and the analysis reaches a fixpoint.

4.4.2 Abstract Expression Evaluation

In Bounded Address Tracking, the abstract values (global, n) for global addresses are basically absolute integers and equal to their concrete values. Thus regular

Unary Operator

$$\widehat{\mathbf{eval}}[\![\odot e]\!](s) := \mathsf{let}(r,o) := \widehat{\mathbf{eval}}[\![e]\!](s)\ \begin{cases} (\mathsf{global}, \mathbf{eval}[\![\odot o]\!](\emptyset)) & \text{if } r = \mathsf{global} \\ (\top_R, \top_{32}) & \text{otherwise} \end{cases}$$

Binary Operator

$$\widehat{\mathbf{eval}}[\![e_1 \odot e_2]\!](s) := \mathsf{let}(r_1,o_1) := \widehat{\mathbf{eval}}[\![e_1]\!](s), (r_2,o_2) := \widehat{\mathbf{eval}}[\![e_2]\!](s)$$
$$\begin{cases} (\mathsf{global}, \mathbf{eval}[\![o_1 \odot o_2]\!](\emptyset)) & \text{if } \odot \text{ not } + \text{ and } r_1 = r_2 = \mathsf{global} \\ (r_1, o_1 + o_2) & \text{if } \odot \text{ is } + \text{ and } r_2 = \mathsf{global} \\ (r_2, o_1 + o_2) & \text{if } \odot \text{ is } + \text{ and } r_1 = \mathsf{global} \\ (\top_R, \top_{32}) & \text{otherwise} \end{cases}$$

Nondet

$$\widehat{\mathbf{eval}}[\![nondet(b)]\!](s) := (\top_R, \top_b)$$

Conditional

$$\widehat{\mathbf{eval}}[\![e_1 ?\, e_2 : e_3]\!](s) := \mathsf{let}(r,o) := \widehat{\mathbf{eval}}[\![e_1]\!](s)\ \begin{cases} \widehat{\mathbf{eval}}[\![e_2]\!](s) & \text{if } (r,o) = (\mathsf{global}, 1_1) \\ \widehat{\mathbf{eval}}[\![e_3]\!](s) & \text{if } (r,o) = (\mathsf{global}, 0_1) \\ \widehat{\mathbf{eval}}[\![e_2]\!](s) \sqcup \widehat{\mathbf{eval}}[\![e_3]\!](s) & \text{otherwise} \end{cases}$$

Memory

$$\widehat{\mathbf{eval}}[\![m_b[e]]\!](s) := \mathsf{let}(r,o) := \widehat{\mathbf{eval}}[\![e]\!](s)\ \begin{cases} s(\hat{m}_b[r,o]) & \text{if } r \neq \top_R \wedge o \neq \top_{32} \\ (\top_R, \top_b) & \text{otherwise} \end{cases}$$

Table 4.2: Definition of the abstract evaluation operator for Bounded Address Tracking.

arithmetic and bit shifting expressions over them can simply be evaluated concretely over the offset parts of the values, as shown in the first cases of Unary Operator and Binary Operator in Table 4.2. Invocation of the concrete **eval** on an empty state denotes that no variables are assigned for evaluation. Addresses for other, non-global regions (r, n) have no statically known absolute value but correspond to the address $r + n$, where r is symbolic and n is a concrete value. Therefore, additions of positive or negative integers (i.e., global memory addresses) to the offset of such abstract values can be precisely modeled (second and third case of Binary Operator). If pointers to different regions are added or pointers are involved in other types of expressions (including comparisons), the resulting abstract value is safely overapproximated to $(\top_R, \top_{32})$ (second case of Unary Operator and fourth case of Binary Operator).

Explicit nondeterminism in expressions evaluates to the top element of the requested bit length, e.g.., $(\top_R, \top_{32})$ for $nondet(32)$ (rule Nondet). Conditional expressions are evaluated checking whether the conditional guard evaluates to true (1_1) or false (0_1), and return the corresponding subexpression in one of these cases (case 1 and 2 of rule Conditional). If the conditional guard does not evaluate to a definite value, i.e., $(\top_R, \top_1)$ or $(\mathsf{global}, \top_1)$, the values of both expressions are joined with respect to the lattice of abstract addresses (case 3).

Memory reads are interpreted by overapproximating those values from the abstract store that the abstract pointer may point to. In particular, this means that if the region and offset are known (region not $\top_R$ and offset not $\top_{32}$), a single memory location is read (first case of rule Memory). If either is not precisely known, all possible values that are pointed to have to be overapproximated. Since there is no type information available to limit the points-to set, all values in the same region (or all regions, respectively) have to be overapproximated. This is safely done by using the top element for the bit length of the memory read (second case).

The semantics for the bit-level operations that change the bit length of a value are shown in Table 4.3. Their interpretation follows the implementation of the unary and binary operators, i.e., the results are determined concretely if the ab-

Sign Extension

$$\widehat{\mathbf{eval}}[\![\mathsf{sgnex}(w,e)]\!](s) := \mathsf{let}(r,o_b) := \widehat{\mathbf{eval}}[\![e]\!](s) \begin{cases} (r,o_w) & \text{if } r = \mathsf{global} \\ (\top_R,\top_w) & \text{otherwise} \end{cases}$$

Zero Extension

$$\widehat{\mathbf{eval}}[\![\mathsf{zeroex}(w,e)]\!](s) := \mathsf{let}(r,o_b) := \widehat{\mathbf{eval}}[\![e]\!](s) \begin{cases} (r,o_w) & \text{if } r = \mathsf{global} \wedge o_b \geq 0_b \\ (r,(2^b+o)_w) & \text{if } r = \mathsf{global} \wedge o_b < 0_b \\ (\top_R,\top_w) & \text{otherwise} \end{cases}$$

Bit Extraction

$$\widehat{\mathbf{eval}}[\![e@[v:w]]\!](s) := \mathsf{let}(r,o) := \widehat{\mathbf{eval}}[\![e]\!](s) \begin{cases} (r,\mathbf{eval}[\![o@[v:w]]\!](\emptyset)) & \text{if } r = \mathsf{global} \\ (\top_R,\top_{w-v+1}) & \text{otherwise} \end{cases}$$

Table 4.3: Abstract semantics of bit length casting operations in Bounded Address Tracking.

stract values are absolute integers. If they are not, a top element with the correct bit length is returned. Sign extension does not perform any operation other than change the bit length of the variable, as IL variables are interpreted as signed by default. Zero extension, on the other hand, adds only leading zeros even to negative values, which is reflected in the second case of rule Zero Extension. Bit extraction returns a bit-level substring of the offset value, if the memory region of the abstract value is global.

4.4.3 Abstract Post Operator

The semantics of abstract statements are given by the abstract post operator defined in Table 4.4. A register assignment (rule AssignReg) is interpreted con-

AssignReg

$$\widehat{\mathbf{post}}[\![[v := e]^{\ell}_{\ell'}]\!](s) := \left\{ s[v \mapsto \widehat{\mathbf{eval}}[\![e]\!](s)][pc \mapsto \ell'] \right\}$$

AssignMem

$$\widehat{\mathbf{post}}[\![[m[e_1] := e_2]^{\ell}_{\ell'}]\!](s) := \mathsf{let}(r, o) := \widehat{\mathbf{eval}}[\![e_1]\!](s),\ a := \widehat{\mathbf{eval}}[\![e_2]\!](s),\ s' := s[pc \mapsto \ell']$$

$$\begin{cases} \{s'[\hat{m}[r,o] \mapsto a]\} & \text{if } r \neq \top_R \wedge o \neq \top_{32} \\ \{s'[\hat{m}[r,i] \mapsto s(\hat{m}[r,i]) \sqcup a][\ldots] \text{ for all } i \in \mathbb{I}_{32}\} & \text{if } r \neq \top_R \wedge o = \top_{32} \\ \{s'[\hat{m}[r,i] \mapsto s(\hat{m}[j,i]) \sqcup a][\ldots] \text{ for all } j \in R, i \in \mathbb{I}_{32}\} & \text{if } r = \top_R \wedge o = \top_{32} \end{cases}$$

Alloc

$$\widehat{\mathbf{post}}[\![[\mathsf{alloc}\ v, e]^{\ell}_{\ell'}]\!](s) := \left\{ s[v \mapsto (r_s, 0)][pc \mapsto \ell'], \begin{array}{l}\text{where } r_s \text{ is a fresh} \\ \text{region identifier}\end{array} \right\}$$

Free

$$\widehat{\mathbf{post}}[\![[\mathsf{free}\ v]^{\ell}_{\ell'}]\!](s) := \mathsf{let}(r, o) := s(v),\ s' := s[pc \mapsto a]$$

$$\begin{cases} \emptyset \text{ (raise error)} & \text{if } r \in \{\top_R, \mathsf{global}\} \vee o \neq 0_{32} \\ \{s'[\hat{m}[r,i] \mapsto (\top_R, \top_{32})][\ldots] \text{ for all } i \in \mathbb{I}_*\} & \text{otherwise} \end{cases}$$

Assume

$$\widehat{\mathbf{post}}[\![[\mathsf{assume}\ e]^{\ell}_{\ell'}]\!](s) := \begin{cases} \emptyset & \text{if } \widehat{\mathbf{eval}}[\![e]\!](s) = (\mathsf{global}, 0_1) \\ \{s[pc \mapsto \ell']\} & \text{otherwise} \end{cases}$$

Assert

$$\widehat{\mathbf{post}}[\![[\mathsf{assert}\ e]^{\ell}_{\ell'}]\!](s) := \begin{cases} \emptyset \text{ (raise error)} & \text{if } \widehat{\mathbf{eval}}[\![e]\!](s) = (\mathsf{global}, 0_1) \\ \{s[pc \mapsto \ell']\} & \text{otherwise} \end{cases}$$

Havoc

$$\widehat{\mathbf{post}}[\![[\mathsf{havoc}\, v_b <_u n]^{\ell}_{\ell'}]\!](s) := \left\{ s[v \mapsto (\mathsf{global}, i)][pc \mapsto \ell'] \mid i <_u n, i \in \mathbb{I}_b \right\}$$

Table 4.4: Definition of the abstract post operator for Bounded Address Tracking.

cretely and replaces an existing mapping in the new abstract state. Rule AssignMem defines the abstract semantics for an assignment to a memory location, i.e., an assignment to a dereferenced pointer. There are three cases to consider depending on the abstract value of the pointer ((r, o) in the rule).

Strong update: If both region and offset of the pointer are known, a strong update can be performed (first case in rule AssignMem). A strong update allows to replace the old value of the memory location in the abstract store of the new state.

Weak update to a single region: If only the region of the pointer is known and the offset has the abstract value $\top_{32}$, a weak update to the known region only can be performed (second case of AssignMem). Since the precise offset is not known, all memory locations in the region of the abstract store *may* hold the new value after the update, so the existing values have to be joined with the new value (with respect to the lattice of abstract addresses shown in Figure 4.1).

Weak update to all regions: If neither the region nor the offset of the pointer is known, all memory locations in all regions have to be joined with the new value (case 3 of AssignMem). This can be caused by the dereference of an uninitialized pointer or the allocation of memory in a loop whose bound exceeds the value bound.

In practice, the state becomes too imprecise to continue analysis after a weak update to all regions. In particular, all return addresses stored on the stack will be affected by the weak update as well. Therefore, Jakstab signals an error for writing to an unknown (possibly null) pointer in this case, but can also be configured to either continue the analysis or ignore the weak update.

Besides the fact that region and offset have to be known, there is another prerequisite for performing strong updates: The region of the pointer must not be a *summary region*, i.e., on all execution paths, the abstract region corresponds only to one concrete memory region [28]. The analysis never creates summary regions, which can be seen from the Alloc rule in Table 4.4. New regions are unique to the abstract state in which they were created. The only way the abstract region value of a pointer can represent multiple regions is if the number of regions for the pointer exceeds the value bound k and is joined to $\top_R$. In this case, a weak update to all regions will be performed when the pointer is dereferenced, which is a sound abstraction for an assignment to a summary region.

The abstract post operator for free sets all memory locations in the freed region to $(\top_R, \top_{32})$. It signals an error if (i) the pointer being dereferenced points to the global region or the unknown region $\top_R$ or (ii) if the pointer does not point to the base (offset 0) of a valid memory region (first case of rule Free). The abstract semantics for assume and assert are similar to the concrete case and only adapted to the lattice of abstract addresses. The abstract post for havoc is the only implementation that returns a non-singleton set: It splits abstract states by enumerating absolute integer values of the bit length suitable for the given register v up to the supplied value n. The use of havoc allows to precisely represent the outcome of nondeterministic choice; this will be discussed in detail in the following section.

4.5 Abstraction of Nondeterminism

Abstraction by approximating multiple concrete program states with abstract states is the key to achieving scalability of an analysis. In static analysis, abstraction is introduced by choosing a suitable abstract domain for the program to be analyzed. In software model checking, an iterative refinement finds a suitable abstraction by adding new predicates over program variables. Control flow reconstruction from binaries requires concrete values for jump targets, however, and the lack of types requires precise values for pointer offsets. Therefore, existing mechanisms for abstraction are not well-suited for a precise analysis of binaries. Still, abstraction has to be introduced to make the analysis feasible.

Even though Bounded Address Tracking resembles software model checking in the way that states from different paths are not merged, it allows registers and memory locations to be *unknown*, i.e., set to $(\top_R, \top_b)$. This is especially useful when representing nondeterminism in the execution environment (e.g., input, unspecified behavior, etc.). Setting parts of the state to unknown avoids the exponential enumeration of possible value combinations. When designing the environment model for a program, the analyst often has a good idea of what parts need to be precisely modeled and where multiple states can be safely combined.

For instance, the standard calling convention in Windows that is used for most API functions specifies that upon return the contents of registers `eax`, `ecx`, and `edx` is undefined. Enumerating all possible values for the registers in a full explicit state exploration would require creating 2^{96} states. By abstracting the nondeterministic choice of values to the value $(\top_R, \top_{32})$ for all three registers, only a single abstract state is required. It is extremely unlikely to produce a spurious counterexample from this abstraction, since code should not depend on undefined side effects.

On the other hand, there are occasions when abstracting to $(\top_R, \top_{32})$ increases the requirements for the abstract domain. Consider the following code, which is a C language stub for the Windows API function IoCreateSymbolicLink:

```
int choice = nondet32;               mov eax, nondet32
if (choice == 0)                     neg eax
  return STATUS_SUCCESS;             sbb eax, eax
else                                 and eax, 0xC0000001
  return STATUS_UNSUCCESSFUL;        ret
```

Here, the compiler replaced the conditional statement with a bitwise operation. Bounded Address Tracking can only deduce that `eax` is $(\top_R, \top_{32})$ at the return statement, even though it actually can be only either `0` or `0xC0000001`. This is where the usefulness of the havoc statement becomes apparent; it causes the analysis to generate multiple successor states with different integer values for a register (Havoc in Table 4.4). Using havoc, the first line of the stub can be rewritten as `int choice; havoc(choice, 1)`. This causes the analysis to create two states; one with `eax` set to 0, and one with `eax` set to 1. From these states it can easily compute the two possible states at the return statement: In the first case `eax` becomes `0`, in the second case `0xC0000001`.

4.6 Implementation Issues

When implementing an analysis such as Bounded Address Tracking for the x86 architecture, there are two problems that require particular attention: First, the fact that memory accesses do not have to be aligned and can partially overwrite previously stored information, and second, the aliasing of registers of different bit lengths (e.g., `eax, ax, ah, al`). This section elaborates on the choices that were made in Jakstab for implementing the analysis for the x86.

4.6.1 Representing Byte-Addressable Abstract Memory

For the ease of exposition, the semantics of store accesses in Table 4.2 and Table 4.4 is simplified by assuming that values of arbitrary bit length can be stored at a single store address and no memory locations overlap. In the x86 architecture, however, memory accesses can manipulate several bytes at once (up to 16 bytes on a Pentium III using SIMD extensions). Therefore, stored values at more than one location can be affected by a single write access, and values from multiple addresses can be requested by a single read access. A straightforward idea to implement the byte-addressable memory of the x86 architecture is to store only a single byte at each address, i.e., to break down memory reads and writes into single byte accesses that are split and combined using bit masking expressions. This approach works well with integer values, which can be easily manipulated on the bit-level, but is impractical when reading or writing pointers (i.e., values with a region other than global). Pointers have no single concrete value, so the outcome of bit masking operations would have to be stored symbolically.

The implementation in Jakstab uses a different approach. If an abstract value v with a length of b bytes is assigned to a memory location a, the entire value v is mapped to location a. Additionally, references to a are stored at locations $a+1, \ldots, a+b-1$ that record the fact that these locations are occupied. For example, assume that the four-byte value $v = (\mathsf{stack}, -4_{32})$, a pointer to a memory location on the stack, is assigned to the offset 2 within the allocated heap region alloc_0. The mapping $2 \mapsto (\mathsf{stack}, -4_{32})$ is stored in the hash map for region alloc_0, and references to offset 2 are stored in the map for offsets 3, 4, and 5. In a sequential representation, the map then becomes:

			v	2	2	2			
alloc_0	0	1	2	3	4	5	6	7	8

$v = (\mathsf{stack}, -4_{32})$

Now, if the four-byte integer value $w = (\mathsf{global}, \mathtt{0x20C0FFEE})$ is assigned to offset 4 in the same heap region, the reference to 2 stored at 4 will show that the byte is already occupied by another value. The existing mapping of $2 \mapsto v$ and its references thus become invalidated and every byte is set to the unknown value $(\top_R, \top_8)$ (empty cells in the diagram):

					w	4	4	4	
alloc_0	0	1	2	3	4	5	6	7	8

$w = (\mathsf{global}, \mathtt{0x20C0FFEE}_{32})$

The stored value is an integer, so it is possible to calculate the results of partial read accesses to it by using bit masking. For instance, if a two-byte memory read from offset 6 is interpreted, the implementation reads the reference to offset 4 from the hash map. The read is two bytes off and x86 is a little-endian architecture, so the implementation then extracts the two most significant bytes from w and returns $(\mathsf{global}, \mathtt{0x20C0}_{16})$.

Representing the abstract store explicitly in hash maps is very memory intensive if every abstract state is to have its own exclusive copy of the maps. Therefore the implementation uses a lazy copying scheme with reference counting, such that an abstract state shares most to all hash maps with its predecessor. When a memory location is updated, the reference count tells whether the state already has a unique copy of the map. If not, the map for that particular memory region is copied, the reference in the abstract state is redirected to the new map, and the reference counter of the old map is decreased.

4.6.2 Register Aliasing

A similar problem arises for the 32 bit registers in the x86 architecture (e.g., `eax`, `ebx`). For each of these, there is a 16 bit register (e.g., `ax`, `bx`) that occupies the two least significant bytes and can be used as an operand for 16 bit instructions. For the general purpose registers, the 16 bit part is again split into two 8 bit registers, which allow to individually address the two least significant bytes as, e.g., `ah` (`ax` high byte) and `al` (`ax` low byte). A simple straightforward approach is again to translate accesses to the aliasing registers by bit masking the corresponding 32 bit registers. This suffers from the same imprecisions as in the memory representation, however. Consider the following code:

```
mov eax, dword ptr [ebx]
mov ah, 0xEE
movzx ecx, ah
```

The code moves an unknown value from memory into `eax`, overwrites the 8 bit register `ah` by a constant, and then stores the contents `ah` in the 32 bit register `ecx` by zero extension. It is easy to see that at the end of this code, `ecx` contains the value `0xEE`. A translation to the IL using bit manipulation operations yields the following code:

$$\begin{aligned}&eax := m_{32}[ebx]\\&eax@[8:15] := \texttt{0xEE}\\&ecx := \text{zeroex}(32, eax@[8:15])\end{aligned}$$

Now, if this code is analyzed by Bounded Address Tracking, it can only establish that the abstract value for `eax` after the update in the second line stays at to

$(\top_R, \top_{32})$, since bits 0 to 7 and 16 to 31 of `eax` are unknown. Therefore, nothing is known for `ecx` in the end.

The implementation in Jakstab uses a defensive strategy for resolving register aliasing that attempts to keep values for subregisters intact. On an update to the subregister s of a larger register r with the new value v, the current value of r and the value v are checked:

1. If both are integer values $(\mathsf{global}, \cdot)$, then the update is performed as in the naive bit extraction strategy described above. Since both values are absolute integers, the result can be easily determined.

2. If the value of r is not an integer (i.e., a pointer or $\top_{32}$), then r is removed from the state (i.e., set to $(\top_R, \top_{32})$), and a new mapping $s \mapsto v$ is added to **Val**.

3. If only the value of r is an integer, then this value is split up between the other subregisters besides s using bit masking. Then a new mapping for each of these subregisters and the mapping $s \mapsto v$ are added to **Val**.

Whenever a register is written to (e.g., `eax`), all subregisters (e.g., `ax, ah, al`) are removed from the state again. Read accesses to a subregister first look for a value of the subregister itself in **Val**. If none is stored, the parent registers are checked for an integer value from which the relevant bits can be extracted. Finally, if also no parent register is stored, the smaller registers (if any) are checked for integer values that can be combined to a larger value.

Using this strategy, the code from the above example is analyzed as follows, with register valuations shown on the right:

$$\begin{array}{ll}
eax := m_{32}[ebx] & \emptyset \\
ah := \texttt{0xEE} & (ah \mapsto \texttt{0xEE}) \\
ecx := \text{zeroex}(32, ah) & (ah \mapsto \texttt{0xEE}, ecx \mapsto \texttt{0x000000EE})
\end{array}$$

Since the mapping for `ah` is stored even though the value of `eax` is unknown, the analysis can now deduce the correct value for `ecx` at the end.

4.7 Related Work

Bounded Address Tracking is inspired by the explicit analysis of Beyer et al. [18], which tracks concrete values of integer variables of C programs up to a certain bound. In their work, explicit analysis is used for cheap enumeration of values for a variable before it is modeled by the computationally more expensive predicate abstraction. Bounded model checking for software, as implemented in the *C Bounded Model Checker* (CBMC) [41], bears some resemblance to the approach presented here. CBMC models variables with bit precision and supports pointer arithmetic. In bounded model checking, loops are unrolled up to a certain depth, so absence of assertion failures is only proven up to a given search depth. In Bounded Address Tracking, loops are unrolled until a fixpoint is reached, while termination (with a sound result) is ensured by the bound on the number of values.

Traditionally, heap-aware program analysis tries to prove invariants over the shape of heap structures [121, 122]. Usually, heap objects are assumed to be records that are explicitly accessed, and pointer arithmetic is ignored or overapproximated. A number of source analyses do handle pointer arithmetic, however. The static assertion checker for low level C code HAVOC [29] uses a memory model related to Bounded Address Tracking. In their model, every pointer value consists of two components, an object reference and an offset. Except for the unavailability of types in executable analysis, identifiers for memory regions are similar as object references. Separation [23, 119] is another formalism able to deal with pointer arithmetic. It's model of the heap deals with numeric pointers and offsets, and assumes that different heap objects never overlap (similar to the partitioned memory model in Section 4.2).

As pointed out already, the CodeSurfer/x86 project is most closely related to this work and faces similar challenges. The major differences in approach are that CodeSurfer/x86 is implemented on top of the heuristics based IDA Pro, and that its analyses (in particular Value Set Analysis (VSA) [7]) are based on more "classic" static analyses such as interval analysis. VSA is path insensitive and thus requires the use of call strings for reasonable results. Call strings, however, are tied to the concept of procedures (which is unreliable in x86 assembly) and assume the existence of a separate call stack. This issue lead to the design of the path sensitive analysis presented in this dissertation.

Balakrishnan and Reps generally rely on summary nodes for representing heap objects. They reduce the number of weak updates by introducing a *recency abstraction* [9] of heap nodes. Their approach extends the common paradigm of using one summary node per allocation site (i.e., address of the call to malloc), by splitting this summary node into (i) the region most recently allocated in the current execution path and (ii) a summary node for the remaining regions. In contrast, Bounded Address Tracking instead explicitly discriminates allocated regions up to the value bound.

Implementation-wise, CodeSurfer/x86 uses a similar strategy as Jakstab to reduce the memory usage of abstract states. Instead of lazy copy-on-write hash maps, they use applicative dictionaries, i.e., AVL trees that copy only the modified subtree on an update [104]. Applicative trees have even lower memory requirements; they incur a higher runtime cost of $\mathcal{O}(n \log n)$ for read and write accesses compared to $\mathcal{O}(n)$ for hash maps, however.

Chapter 5

Disassembly and Static Analysis with Jakstab

This chapter presents the architecture and ideas behind Jakstab, a modular and extensible disassembly and static analysis platform for binaries. Jakstab has been developed in Java (about 40 KLOC at the time of writing) and implements the general framework for control flow reconstruction from binaries introduced in Chapter 3. Statements are retrieved from an executable by disassembling instructions on demand and translating them into the IL. By exploring the reachable states of the program, Jakstab can verify assertions while reconstructing the control flow graph on the fly. Similar to the theoretical framework, it is not fixed in its choice of abstract domains; it includes an implementation of Bounded Address Tracking, but also allows to combine various other analyses from the literature and provides a clean API to add new ones.

5.1 General Architecture

Jakstab uses a disassembly dictionary and an instruction specification to map machine code bytes to IL statements, which in turn are used to compute the abstract transfer relation between abstract states. This section details the architectural concepts of Jakstab and its analysis machinery.

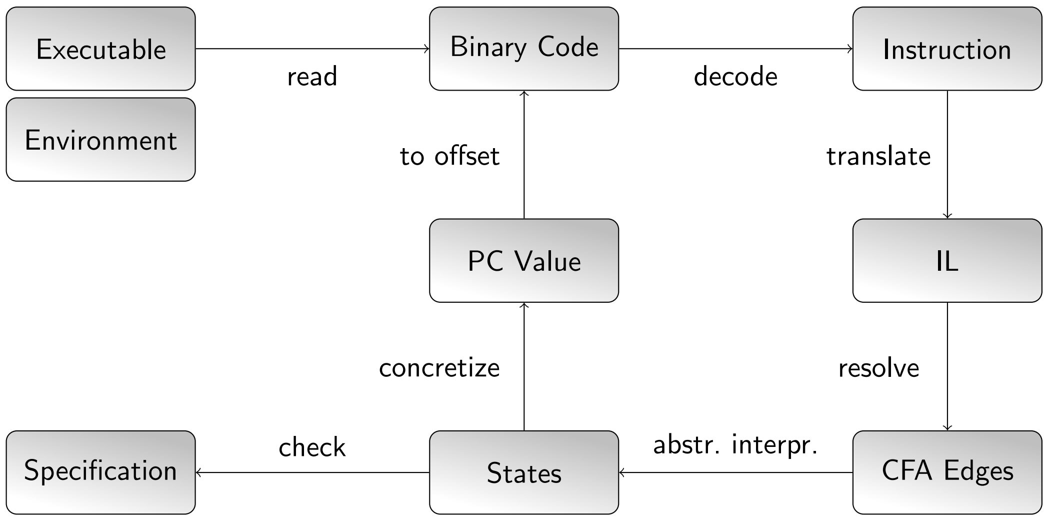


Figure 5.1: Unified disassembly and analysis architecture.

5.1.1 Single Pass Disassembly and Analysis

Existing approaches to static analysis of binary executables rely on a preprocessing step performed by a dedicated, heuristics based disassembler such as IDA Pro [70] to produce a plain text assembly listing [7, 31, 34, 95]. This decouples the analysis infrastructure from disassembly itself and makes it difficult to use results from static analysis towards improving the control flow graph. If the analysis builds on an external disassembler, soundness can only be guaranteed with respect to the (error prone) output produced by the disassembler.

To overcome this problem, Chapter 3 introduced a framework for control flow reconstruction, which in the following will be cast into an architecture for single pass disassembly and analysis of binaries. It does not discriminate between disassembly and analysis stages but integrates both into the same analysis loop (Figure 5.1). The integrative design of the analysis architecture is based on the following key insight: Tracing the control flow of a binary in order to decode the executed instructions is already an analysis of reachable locations. This is non-trivial in presence of indirect control-flow and should not be left to heuristic algorithms.

Using the entry point of the executable as the initial program counter (*pc*) value, Jakstab reads and decodes one instruction at a time from the file offset that corresponds to *pc*. A disassembly logic determines how many bytes to process until the instruction including all operands is completely read. The instruction is then translated into one or more IL statements according to an architecture-dependent instruction definition file. In the next step, the IL statements are resolved into a set of CFA edges by invoking the **resolve** operator (see Section 3.4.1) with data flow information provided by the abstract domain. Most statements yield only a single edge to their fall-through successor; guarded jumps can translate to multiple edges depending on the abstract values for the branch condition and target expression. For each of these CFA edges, Jakstab calculates the set of successor states by interpreting the abstract semantics of the edge's IL statement. The abstract semantics are given either by a single abstract domain or by composing multiple domains.

If one of the new abstract states is an error state with respect to the properties being checked (if any), Jakstab outputs an abstract error trace leading to the state. Due to abstraction, error traces are not necessarily feasible. If no property is violated, the analysis continues: Jakstab concretizes program counter values from the new abstract states, and reads the next instructions to be interpreted from the file offsets corresponding to the new *pc* values. The process continues until all reachable states have been explored or a property violation was found. To guarantee termination in presence of loops, the abstract domain may have to include a widening operator, such as the enforcement of value bounds in Bounded Address Tracking (see Section 4.4).

Overall, this amounts to abstract interpretation of a system comprised of the processor registers and virtual memory, the actual binary program, and the CPU's instruction fetch. For performance reasons, instructions are read directly from the file instead of preloading the file to memory. This is no conceptual limitation, however, and by preloading the binary to memory and decoding instructions from the abstract memory representation, the same approach can be used for analyzing self-modifying code.

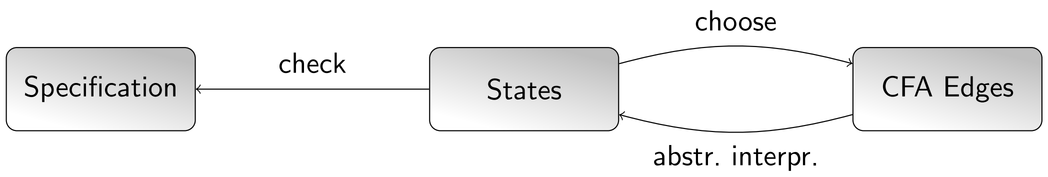


Figure 5.2: Secondary analysis performed on the reconstructed CFA.

5.1.2 Secondary Post-Reconstruction Analysis

The architecture described above includes property checking as part of the control flow reconstruction. Indeed, since the abstract domain for control flow reconstruction has to be very precise and the analysis performs an exhaustive state space exploration for the given abstract domain, property checking can often be performed as a "byproduct" of control flow reconstruction. Property checking then simply amounts to finding error states within the program's abstract state space. Still, it is possible to run another, unrelated analysis on the reconstructed control flow automaton, completely separate from control flow reconstruction (see Figure 5.2). In case the second analysis cannot contribute to control flow reconstruction (i.e., it does not assist in calculating precise addresses) this separation of concerns can improve performance. Furthermore, the control flow graph can be transformed between both analysis phases, as will be discussed in Section 5.4.

Besides property checking, such a second analysis phase can be used for other, classic program analysis scenarios such as detecting dead code, or identifying dependencies between program parts. A practical use of this second phase implemented in Jakstab was its use as a platform for teaching a program analysis lab course during two terms at Technische Universität Darmstadt [71]. The second analysis phase invoked program analyses developed by students without interfering with control flow reconstruction. Intermediate CFA simplification and expression substitution can remove many of the difficulties associated with analyzing binaries and allow students to work with binaries similarly as they would with higher level programs.

5.1.3 Program Representation

Jakstab uses three different concepts to represent and reason about programs: *Instructions*, *IL statements*, and *CFA edges*. Each belongs to a particular stage of the disassembly and analysis process shown in Figure 5.1. On the lowest level, there is a mapping from addresses to machine-dependent assembly instructions. These are translated into the lower intermediate representation, a mapping from *labels* to IL statements. By determining the control flow targets of these statements, the analysis constructs the control flow automaton of the program, which forms the highest level of program representation. All three concepts are present while the analysis runs. Figure 5.3 shows a piece of assembly code (from the example in Section 3.2) in all three representations. Assembly instructions are obtained by disassembly from the binary, IL statements are generated from translating the assembly instructions, and CFA edges are resolved by analyzing IL statements.

Instructions. The disassembly logic decodes one or more bytes at the location pointed to by the program counter into one assembly instruction. Every instruction is stored in the *instruction map* from virtual addresses to instruction objects. Jakstab represents instructions as compound objects of several types for a particular architecture (e.g., move instructions, jump instructions) consisting of an opcode and a list of operands that again are of some architecture dependent type (e.g., register, memory operand).

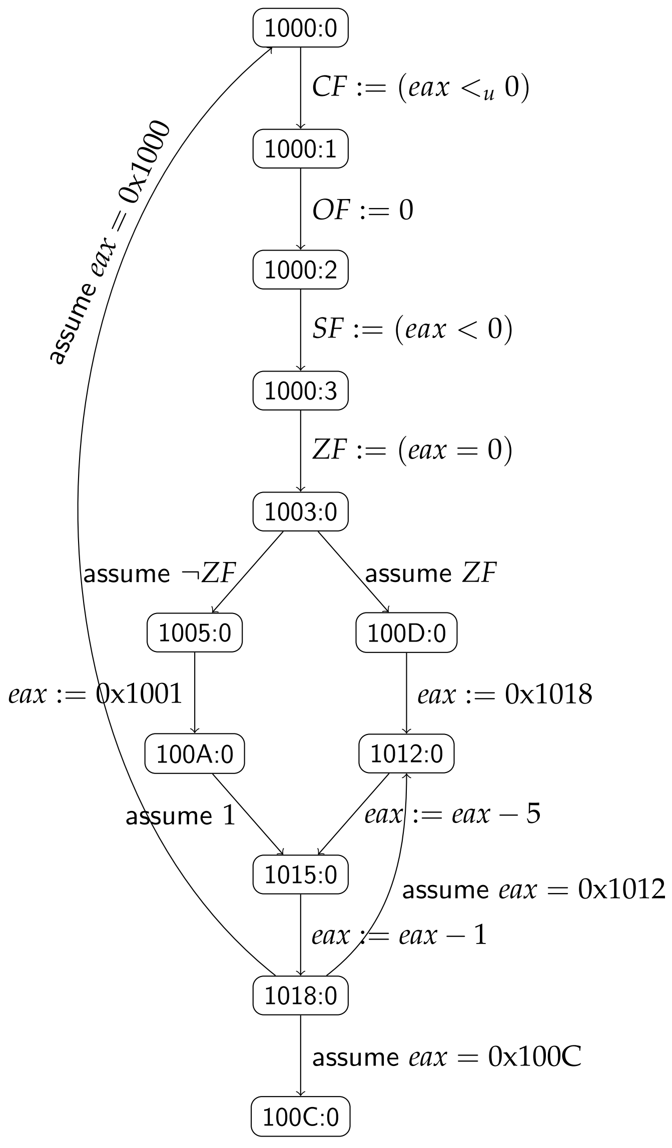
The class hierarchy of instructions in Jakstab is based on OpenJDK's [111] disassembler architecture, which resides in the `sun.jvm.hotspot.asm` package. It is capable of supporting multiple architectures by offering abstract classes for common instruction types which are extended by implementation for particular architectures. Currently, only the x86 architecture is supported in Jakstab, but other architectures can be integrated by adding the corresponding implementation classes and a translation table from opcodes to instruction objects.

```
0x1000  cmp   eax, 0
0x1003  jz    0x100D
0x1005  mov   eax, 0x1001
0x100A  jmp   0x1015
0x100C  retn
0x100D  mov   eax, 0x1018
0x1012  sub   eax, 5
0x1015  sub   eax, 1
0x1018  jmp   eax
```

(i)

1000:0:	$CF := (eax <_u 0)$
1000:1:	$OF := 0$
1000:2:	$SF := (eax < 0)$
1000:3:	$ZF := (eax = 0)$
1003:0:	if ZF jmp 0x100D
1005:0:	$eax :=$ 0x1001
100A:0:	if 1 jmp 0x1015
100C:0:	halt
100D:0:	$eax :=$ 0x1018
1012:0:	$eax := eax - 5$
1015:0:	$eax := eax - 1$
1018:0:	if 1 jmp eax

(ii)



(iii)

Figure 5.3: The three levels of program representation in Jakstab: (i) map from virtual addresses to assembly instructions, (ii) map from labels to IL statements, (iii) control flow automaton. Flag updates after both `sub` instructions have been left out for simplicity.

Statements. Using an SSL definition file (see Section 2.6), every assembly instruction is decoded into a sequence of basic IL statements (register and memory assignments, guarded jumps, halt). The SSL file contains parameterized templates of IL statements for the available opcodes. For each disassembled instruction, Jakstab looks up the corresponding template for the instruction's opcode and instantiates the template with the instruction operands. If the program counter is referenced in the template, it is instantiated with the address immediately following the current instruction, which is the runtime value of the program counter on x86 architectures.

A single instruction at a single virtual address can translate to multiple IL statements, thus a single virtual address is split into multiple *labels*: A label consists of a virtual address combined with an index that uniquely identifies a statement in the program. All statements that have been translated from instructions are stored in a global *statement map* from labels to IL statements. The statement map does not store the resolved assume statements but only the original guarded jumps.

Control Flow Edges. The advantage of a CFA over the classical control flow graph (CFG) is that vertices naturally correspond to states at a particular program location, with edges representing state transformers. Furthermore, a CFG only allows one statement (or basic block) per location, while the CFA can have different state transformers originating from the same location. Thus the CFA lends itself particularly well towards control flow reconstruction, where a jump is transformed into multiple assume statements.

CFA edges are produced by resolving IL statements (see Section 3.4.1) and are stored as a set during control flow reconstruction. After the initial control flow reconstruction finishes, the set forms the CFA and can be used for a secondary analysis as described in Section 5.1.2. The resolved CFA contains no jump statements but fully encodes the semantics of the IL code derived from the executable.

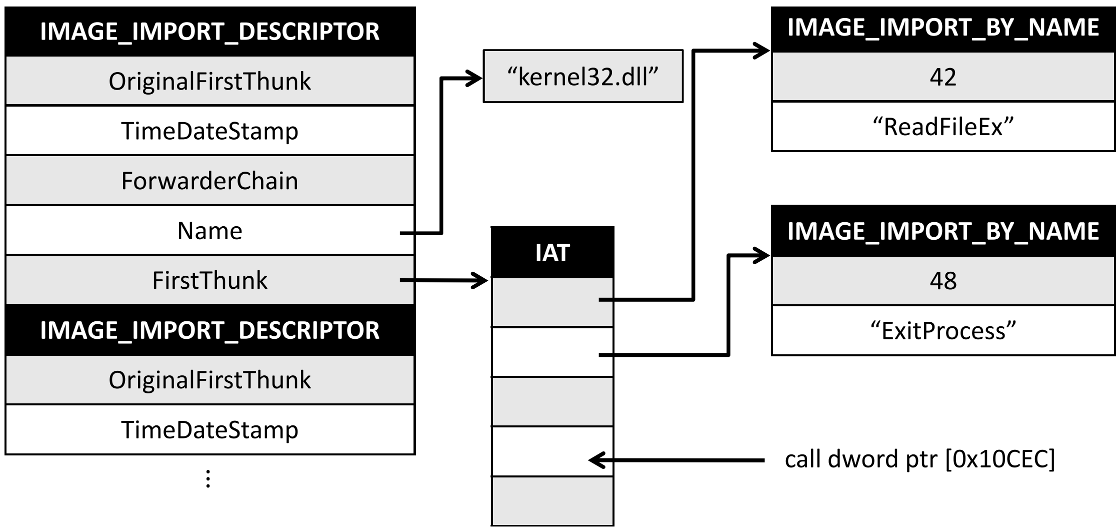


Figure 5.4: Dynamic linking in Windows PE files. The loader overwrites the Import Address Table (IAT) with the correct function entry points (adapted from [112]).

5.1.4 Execution Environment

An executable relies on the operating system for several critical initialization functions, which have to be simulated by a static analysis tool in order to correctly predict the actual runtime behavior.

Dynamic Linking. The theoretical framework treats procedure calls (`call` instructions) in the same way as unstructured gotos (`jmp` instructions). Regular Windows programs contain calls to routines imported from dynamic link libraries (DLLs), however, and the analysis framework is required to handle them. In Windows PE files, these calls are encoded as indirect calls to a specific entry in the file's *import address table* (IAT). When the executable is run, the loader of the operating system parses a list of import descriptors (one for each library) referenced from the executable header [112]. Each entry references the name of a library and contains a pointer into the IAT (Figure 5.4). The loader then finds the library by its name and maps it into the address space of the new process, before processing the section of the IAT the import descriptor points to. The IAT serves two purposes: Initially, it contains relative file pointers to structures of the type IMAGE_IMPORT_BY_NAME, which hold the name of the function to import and a hint for the loader on where to start a binary search in the referenced library. The loader reads each structure and replaces the IAT pointer to it with a pointer to the runtime address of the imported function in the library. After loading completes, each pointer in the IAT points to the runtime address of the function it imports and the IAT takes up its second duty of redirecting function calls. Calls to imported library functions indirectly reference the entries in the IAT and therefore transfer control to the correct function entry point.

For the static analysis to correctly handle imports, the interaction of the executable with the *execution environment* provided by the operating system has to be modeled, i.e., the functionality of the loader has to be integrated into the analysis. Before the first instructions are disassembled, Jakstab thus finds the import table, reads the names of referenced routines and libraries, and enters valid addresses for the imported routines. There are four different ways the imports can be resolved:

1. If the imported DLL is explicitly specified by the user to be analyzed in conjunction with the main executable, the imported routines are resolved to the corresponding exported function entry points in the DLL. This is the most precise way of handling imports, but including all DLLs belonging to the Windows API leads to an unacceptable increase in size of the analyzed code.

2. It is possible to replace the referenced DLL by an abstracted, simplified version. To this end, the user creates a new library exporting the functions to be abstracted, and loads it alongside the main executable. Jakstab then enters the addresses of the replacement routines into the import table. This method is used for abstracting many library functions of the Windows driver interface in the case study conducted in Section 6.1.

3. If neither the actual implementation nor a user provided stub are available, Jakstab can parse a *module definition file* for the referenced library to extract at least the calling conventions of the imported routines and automatically create a stub implementation of IL statements in the statement map. The stub attempts to overapproximate the routine's behavior by assigning *nondet* to the registers eax, ecx, and edx, which all of Windows' calling conventions allow to be overwritten by procedures [100]. If the calling convention specifies that the parameters are cleared off the stack by the callee, the stub increases the stack pointer accordingly and then returns.

   Jakstab uses an additional annotation in module definition files that specifies whether the procedure ever returns. For instance, the Windows API function ExitProcess never returns to the calling process but terminates it, so a call to it can be followed by illegal code. An analysis has to be aware that execution does not continue after non-returning functions, so that it does not create spurious control flow. Jakstab adds an explicit halt statement to non-returning stubs for non-returning functions.

4. Finally, if no matching module definition is available or the procedure is missing from it, Jakstab generates a default stub that assumes the standard C calling convention _cdecl in which the caller clears parameters off the stack [100], since the number of parameters is unknown. If the assumption is wrong and the actual calling convention demands the callee to clean the stack, a subsequent return of the calling procedure will likely use the wrong return address from the stack as the stack pointer is now misaligned. Therefore, Jakstab issues a warning whenever it has to create a default stub for imported routines.

Prologue and Epilogue. Another implicit interaction between an executable and the execution environment consists of how the executable's main method is actually invoked. Section 4.3 already briefly mentioned how the initial state for the analysis has to be set up: If a main function does not explicitly exit using

ExitProcess, it simply returns to the operating system. To create a closed program that terminates with an explicit halt statement, a short prologue sequence of IL statements is used as starting point for the analysis:

$$\begin{aligned}&DF := 0_1\\ &esp := esp - 4_{32}\\ &m_{32}[esp] := \text{EPILOGUE\_ADDRESS}\\ &\mathsf{if}\ 1_1\ \mathsf{jmp}\ \text{ENTRY\_POINT}\end{aligned}$$

The first line sets up the value of 0 for the direction flag DF, which is guaranteed by the operating system and specifies the direction of string operations. The second and third lines then push the address of a short epilogue consisting of only a halt statement onto the stack, and the fourth line jumps to the entry point of the executable being analyzed. Prologue and epilogue are inserted into the statement map before analysis starts.

5.2 Modular Implementation of Different Analyses

Implementing a static analysis such that it is correct and fulfills the needs of the specific problem at hand is a time consuming process, so an analysis framework should maximize the reusability of individual abstract domains and analyses. The design of the Jakstab framework was thus driven by the idea of allowing to specify abstract domains individually and leaving the composition of individual domains largely to the framework.

Abstract interpretation already defines ways to build combinations of existing abstract domains [47], but the *Configurable Program Analysis* (CPA) framework defined by Beyer et al. [17] offers a particularly clean separation of concerns and easy modification of existing analyses. This section gives a brief introduction to the CPA framework and explains how it was adapted for use with Jakstab.

5.2.1 Configurable Program Analysis

In [17, 18], the authors introduce *Configurable Program Analysis* (CPA), a framework for systematically configuring and combining analyses from the areas of software model checking and static program analysis. Using a standardized interface, it allows easy combination of different analyses into a composite abstract domain and analysis. The CPA interface factors crucial aspects, which define the nature of an analysis, into a number of operators that can be easily redefined to create analyses of different precision.

CPA uses a standard, generic worklist algorithm that invokes the operators to compute the set of reachable states, which is shown in Figure 5.5. Abstract states are associated with *precision* elements that contain dynamic configuration information for the domain, e.g., which parts of the abstract state are modeled at which granularity. The algorithm maintains a set reached of abstract states with precision that have been explored so far and a worklist of abstract states with precision to explore in the next iterations. An individual analysis $\mathbb{D} = (\mathcal{L}, \gamma, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ is defined by the following components (slightly adapted from [18]):

- A join semilattice $\mathcal{L} = \langle D, \top, \bot, \sqsubseteq, \sqcup \rangle$ of abstract states, with the set of elements D, an upper bound (top element) $\top \in D$, a lower bound (bottom element) $\bot \in D$, the partial order $\sqsubseteq \subseteq D \times D$, and the join operation $\sqcup :: D \times D \to D$.

- A concretization function $\gamma :: D \to 2^S$ from abstract states to sets S of concrete states, which characterizes the connection between the abstract and concrete semantics. For correct CPA formalizations Beyer et al. [18] require that

  1. $\gamma(\top) = S$ and $\gamma(\bot) = \emptyset$
  2. $\forall a, a' \in D.\, a \sqsubseteq a' \implies \gamma(a) \subseteq \gamma(a')$.

  Note that this definition assumes that the concrete semantics corresponds to the forward (reachability) collecting semantics. CPA is not limited to

```
Input: A configurable program analysis with dynamic precision
       adjustment $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$, an initial abstract state
       $a_0 \in A$ with precision $\pi_0 \in \Pi$, where $A$ denotes the set of elements
       of the semilattice of $D$, and a control flow automaton given as the set
       of edges $G$.
Output: A set of reachable abstract states.
Data: A set reached of elements of $A \times \Pi$, a set worklist of elements of $A \times \Pi$.
worklist := $\{(a_0, \pi_0)\}$;
reached := $\{(a_0, \pi_0)\}$;
while worklist $\neq \emptyset$ do
    pop $(a, \pi)$ from worklist;
    // Adjust the precision.
    $(\hat{a}, \hat{\pi}) = \mathsf{prec}(a, \pi, \mathsf{reached})$;
    foreach $a'$ with $\exists g \in G.\ \hat{a} \overset{g}{\rightsquigarrow} (a', \hat{\pi})$ do
        // Combine with existing abstract states.
        foreach $(a'', \pi'') \in \mathsf{reached}$ do
            $a_{new} := \mathsf{merge}(a', a'', \hat{\pi})$;
            if $a_{new} \neq a''$ then
                worklist := $(\mathsf{worklist} \cup \{(a_{new}, \hat{\pi})\}) \setminus \{(a'', \pi'')\}$;
                reached := $(\mathsf{reached} \cup \{(a_{new}, \hat{\pi})\}) \setminus \{(a'', \pi'')\}$;
        // Add new abstract state?
        if $\neg\mathsf{stop}(a', \{a \mid (a, \cdot) \in \mathsf{reached}\}, \hat{\pi})$ then
            worklist := $\mathsf{worklist} \cup \{(a', \hat{\pi})\}$;
            reached := $\mathsf{reached} \cup \{(a', \hat{\pi})\}$;
return $\{a \mid (a, \cdot) \in \mathsf{reached}\}$;
```

Figure 5.5: The CPA+ algorithm for determining the set of reachable states, adapted from [18].

describing analyses that calculate reachability properties, however. In fact, forward expression substitution and live variable analysis will be formalized as CPAs in Section 5.3. In a slight deviation from the CPA framework, γ will be described as mapping to the collecting semantics for the property of interest instead of the collecting semantics of reachable states in these cases.

- A set of precisions Π, which store information about the granularity with which states are tracked and by which analysis. Not all analyses make use of precision refinement; where it is not used, the set of precisions can be chosen to be a singleton set containing only a *null* element, i.e., $\Pi = \{null\}$.
- The abstract transfer relation $\rightsquigarrow \subseteq (D \times G \times D \times \Pi)$ encodes the abstract semantics for the domain. It relates each state to a set of successor states with precision that are reachable via a given edge of the CFA. The CFA is represented as the set of edges G.
- The precision adjustment function $\mathsf{prec} :: (D \times \Pi \times 2^{D \times \Pi}) \mapsto (D \times \Pi)$ manages which parts of the state are modeled and by which subanalysis. For an abstract state, a precision, and a set of reached states and precisions, it calculates an adjusted abstract state and precision.
- The merge operator is invoked after calculating the successors of a state in the transfer relation. It attempts to merge the new state with states that have been reached before, and can be used to implement joining analyses in the spirit of classical data flow algorithms.
- The operator stop decides whether the new state adds information in comparison to the previously reached states and should be added to the worklist to be explored further.

When combining different analyses, the operators for the composite analysis can be created from the implementations of the individual analyses. For a simple direct product of domains, the composite operators can call the individual implementation independent of each other [17]. The direct product domain can be too imprecise [42, 47]; improved combinations of domains such as the *reduced product* [47] or the *logical product* [64] offer higher precision by exchanging information between domains. To facilitate such improved combinations, the composite transfer relation can be modified to invoke a *strengthening* function for each component of the successor state that incorporates information from other components. Thus each analysis can incorporate information from other abstract states to achieve improved precision. Besides using the interface for strengthening, the composite operators can be overridden for full flexibility.

5.2.2 Modifications to the Worklist Algorithm

Jakstab's analysis architecture is designed in terms of the CPA framework and allows to define analyses providing only the CPA operators. Its analysis algorithm (see Figure 5.6) modifies the CPA worklist algorithm to accommodate the needs of control flow reconstruction.

Resolve Operator. The original CPA algorithm takes an abstract state from the worklist and enumerates all its successor states directly reachable via all feasible CFA edges. Therefore, the algorithm implicitly assumes the CFA to be available, which is not true for binaries before control flow reconstruction. The modified Jakstab algorithm, on the other hand, is able to reconstruct the CFA at runtime using *transformer factories*.

The modified algorithm no longer takes the CFA as input, but instead uses the (implicit) statement map from program locations (addresses) to IL statements. The map is implemented by disassembling yet unexplored instructions on the fly, translating them to IL statements, and storing them in the map as described in Section 5.1. In Line 7, the Jakstab algorithm calls the transformer factory function getTransformers, which produces edges from the current abstract state and the statement map. For each of these new edges, the algorithm computes the set of successor states with respect to the transfer relation. Any new edges are

```
Input: A CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$, an initial abstract state
       $a_0 \in A$ with precision $\pi_0 \in \Pi$, where $A$ denotes the set of elements
       of the semilattice of $D$, and a map $P$ from locations to statements.
Output: A control flow automaton for $P$ and the set of reachable states.
Data: A map reached from program locations to elements of $A \times \Pi$, a set
      worklist of elements of $A \times \Pi$.
worklist := $\{(a_0, \pi_0)\}$;
reached$(a_0(pc))$ := $\{(a_0, \pi_0)\}$;
while worklist $\neq \emptyset$ do
    pop $(a, \pi)$ from worklist;
    // Adjust the precision.
    $(\hat{a}, \hat{\pi}) = \mathsf{prec}(a, \pi, \mathsf{reached})$;
    foreach $g \in \mathsf{getTransformers}(\hat{a}, P)$ do
        foreach $a'$ with $\hat{a} \overset{g}{\rightsquigarrow} (a', \hat{\pi})$ do
            $G := G \cup g$;
            // Combine with existing abstract states.
            foreach $(a'', \pi'') \in$ reached do
                $a_{new} := \mathsf{merge}(a', a'', \hat{\pi})$;
                if $a_{new} \neq a''$ then
                    worklist := (worklist $\cup \{(a_{new}, \hat{\pi})\}) \setminus \{(a'', \pi'')\}$;
                    reached := (reached $\cup \{(a_{new}, \hat{\pi})\}) \setminus \{(a'', \pi'')\}$;
            // Add new abstract state?
            if $\neg\mathsf{stop}(a', \{a \mid (a, \cdot) \in \mathsf{reached}\}, \hat{\pi})$ then
                worklist := worklist $\cup \{(a', \hat{\pi})\}$;
                reached := reached $\cup \{(a', \hat{\pi})\}$;
return $G, \{a | (a, \cdot) \in \mathsf{reached}\}$;
```

Figure 5.6: The Jakstab algorithm, a control flow resolving version of the CPA+ algorithm.

stored in the set G, thus growing the control flow graph in the same manner as the generic worklist algorithm (see Figure 3.3). After the algorithm finishes, the CFA is returned as part of the algorithm's output.

The algorithm can be configured by modifying the transformer factory, i.e., the getTransformers method. For control flow reconstruction, the method has to implement the **resolve** operator (see Section 3.4.1) to translate IL statements to edges. To this end, Jakstab's interface for abstract states requires an evaluation method (the abstract $\widehat{\mathbf{eval}}$ operator in the framework), which concretizes the results of abstractly evaluating jump expressions. The implementations of a resolving getTransformers method can call this evaluation method for retrieving sets of possible concrete target addresses. Heuristics based approaches, as will be discussed in Section 5.2.3, can be implemented through using transformer factories that make optimistic assumptions about control flow if the data flow information is imprecise. For running secondary analyses on the reconstructed CFA (see Section 5.1.2), the method can be replaced by a version that simply looks up outgoing CFA edges from the current program location. The algorithm then becomes the non-resolving standard version of CPA.

5.2.3 Balancing Soundness and Coverage

So far, the work in this dissertation has focused on achieving a high precision analysis that avoids spurious control flow and is robust against unconventional control flow introduced by compiler optimizations or deliberate obfuscation. In many scenarios, however, it can be acceptable to introduce unsoundness into the analysis if a precise analysis is infeasible.

One particular problem are calls to unknown external functions that are supposed to clean parameters off the stack (see Section 5.1.4). In this case, the correct stack height (the value of the stack pointer relative to its initial value upon entering the current function) is lost, causing erroneous updates to local variables and a failing return at the end of the current function.

Resolving Pessimistic. The default transformer factory in Jakstab implements the **resolve** operator by generating edges from the available data flow information. It is *pessimistic* in that it makes no assumptions about well-behavedness of procedures and treats calls and returns exactly like jumps. The advantage is that the analysis becomes resistant against obfuscation techniques that modify the call stack to abuse return instructions for arbitrary control flow. Furthermore, it avoids the common problem of illegal code following a call to a procedures that never returns, e.g., because it calls ExitProcess.

The implementation of the resolving pessimistic transformer factory differs from a fully strict implementation of **resolve** in that it does not generate edges to *all* addresses in the executable if no information about the target address is available. Instead, it signals an error and outputs an error trace. Jakstab can be configured to ignore this problem, treat the failing branch instruction as a halt, and to continue with the analysis on other states in the worklist. This renders the result of the analysis unsound, but allows to explore some more states and gives a chance to find property violations elsewhere.

Resolving Semi-Optimistic. A more optimistic approach is to assume that calls whose target cannot be determined statically are well-behaved and return to the instruction following the call. The unresolved procedure call is replaced by a generic stub, similar to those for imported functions discussed in Section 5.1.4 . If the procedures possibly called in all concrete executions are indeed well-behaved and the stub defines a sound overapproximation of all the procedure's side effects, an analysis can still produce a sound result.

Typically, this transformer factory is going to be used if soundness is less important, though, and stubs can use rough approximations that nondeterministically update registers according to the default calling conventions. The problem still remains whether parameters are cleared off the stack by the caller or the callee, so this method will lead to bad stack pointer information for calls to procedures not following the _cdecl calling convention.

Resolving Optimistic. When sound analysis is unnecessary and only coarse abstract domains such as constant propagation are used, the optimistic transformer factory can add *two* edges for each function call:

- A fall-through edge to the return location of the call that approximates side effects of the procedure in the same way as the generic stubs for unknown calls in the semi-optimistic transformer factory.
- An edge to the target of the call, if it is known.

This causes control flow to branch out on every call. The transformer factory generates halt statements from return instructions, so every procedure eventually ends in a leaf.

With this configuration, Jakstab essentially becomes a classic recursive traversal disassembler similar to IDA Pro, which follows a best effort approach to disassemble as many instructions as possible without being concerned with soundness. To increase coverage of the disassembly further, this configuration can be coupled with heuristics for identifying procedure entry points.

Non-resolving Forward. A secondary analysis, which is run after the CFA has been fully reconstructed, can be implemented by choosing a non-resolving transformer factory. It is initialized with the complete CFA and looks up outgoing edges from the CFA location corresponding to the current program counter value. It does not create new transformers but only returns already existing edges from the CFA. With a non-resolving forward transformer factory, the Jakstab algorithm becomes equivalent to the original CPA worklist algorithm.

Non-resolving Backward. Backward analyses such as *live variable analysis* (see Section 5.3.7) can be implemented with a non-resolving backward transformer factory: Dually to the forward transformer factory, it looks up incoming CFA edges for the location the current abstract state is associated with. If the CFA has multiple exit nodes (nodes without outgoing edges), a virtual location that is connected to all exit nodes (by a dummy statement such as assume 1_1) can serve as initial location for the backward analysis.

5.2.4 Composite Analysis with Selective Merging

In general, composite CPAs can be custom defined to create a suitable composition of individual analyses. By default, however, Jakstab composes analyses in a generic manner that does not require individual customizing. The transfer relations for the component analyses are calculated separately, and the composite transfer relation is constructed as the Cartesian product of successors along the same CFA edge. If available, the implementation calls strengthening operators that can exchange information between analysis components.

The merge and stop operators are executed componentwise except that they keep composite states separate if specifically selected state components (e.g., location or calling context) are incomparable. Selectively merging and stopping only states with equal location components leads to flow sensitive analysis, and selectively merging and stopping only states with equal calling context (call stack) components leads to a context sensitive analysis. The composite CPA is thus parameterized by n individual CPAs $\mathbb{A}_i$ and an index $0 \leq h \leq n$ that identifies those analyses $\mathbb{A}_{0 \leq h}$ which have been selected to prevent joining of components. If $h = 0$, no state splitting is enforced. The composite CPA is defined as $\mathbb{D} = (\mathcal{L}_{\mathbb{D}}, \gamma_{\mathbb{D}}, \Pi_{\mathbb{D}}, \rightsquigarrow_{\mathbb{D}}, \mathsf{merge}_{\mathbb{D}}, \mathsf{stop}_{\mathbb{D}}, \mathsf{prec}_{\mathbb{D}})$, where

1. $\mathcal{L}_{\mathbb{D}} = \langle D_{\mathbb{D}}, \top_{\mathbb{D}}, \bot_{\mathbb{D}}, \sqsubseteq_{\mathbb{D}}, \sqcup_{\mathbb{D}} \rangle$ is the product lattice of the n individual join semilattices $\mathcal{L}_{\mathbb{A}_i} = \langle D_{\mathbb{A}_i}, \top_{\mathbb{A}_i}, \bot_{\mathbb{A}_i}, \sqsubseteq_{\mathbb{A}_i}, \sqcup_{\mathbb{A}_i} \rangle$ with
   - the product set of lattice elements $D_{\mathbb{D}} = D_{\mathbb{A}_1} \times \ldots \times D_{\mathbb{A}_n}$,
   - $\top_{\mathbb{D}} = (\top_{\mathbb{A}_1}, \ldots, \top_{\mathbb{A}_n})$,
   - $\bot_{\mathbb{D}} = (\bot_{\mathbb{A}_1}, \ldots, \bot_{\mathbb{A}_n})$,
   - the Cartesian ordering

$$(a_1, \ldots, a_n) \sqsubseteq_{\mathbb{D}} (a'_1, \ldots, a'_n) :\Longleftrightarrow a_i \sqsubseteq_{\mathbb{A}_i} a'_i, 1 \leq i \leq n,$$

- the join operator defined by

$$(a_1, \ldots, a_n) \sqcup_{\mathbb{D}} (a'_1, \ldots, a'_n) = (a_1 \sqcup_{\mathbb{A}_1} a'_1, \ldots, (a_n \sqcup_{\mathbb{A}_n} a'_n).$$

2. $\gamma_{\mathbb{D}}(a_1, \ldots, a_n) = \bigcap_{1 \le i \le n} \gamma_{\mathbb{A}_i}(a_i)$ is the composite concretization function.
3. the product set $\Pi_{\mathbb{D}}$ of composite precisions is defined as $\Pi_{\mathbb{A}_1} \times \ldots \times \Pi_{\mathbb{A}_n}$.
4. the transfer relation calculates successors separately, but allows states to exchange information using strengthening operators of the type $\downarrow_i :: D_{\mathbb{A}_1} \times \ldots \times D_{\mathbb{A}_n} \times \Pi_{\mathbb{A}_i} \to D_{\mathbb{A}_i}$. Strengthening is currently not used by the abstract domains implemented in Jakstab and the default strengthening operator simply returns an unmodified state, but this mechanism allows to define improved interaction between CPAs if required. Formally, the transfer relation is defined as

$$(a_1, \ldots, a_n) \overset{g}{\rightsquigarrow}_{\mathbb{D}} (a''_1, \ldots, a''_n), (\pi_1, \ldots, \pi_n) :\Longleftrightarrow$$
$$\bigwedge_{1 \le i \le n} a_i \overset{g}{\rightsquigarrow}_{\mathbb{A}_i} (a'_i, \pi_i) \wedge \downarrow_i a'_0, \ldots, a'_n, \pi_i = a''_i.$$

   By this definition, there is a composite successor state via edge g only if every analysis component has a successor via g for its component state, i.e., an edge is only feasible in the composite analysis if it is feasible in all component analyses.

5. the selective componentwise merge is defined as

$$\mathsf{merge}_{\mathbb{D}}((a_1, \ldots, a_n), (a'_1, \ldots, a'_n), (\pi_1, \ldots, \pi_n)) =$$
$$\begin{cases} (\mathsf{merge}_{\mathbb{A}_1}(a_1, a'_1, \pi_1), \ldots, \mathsf{merge}_{\mathbb{A}_n}(a_n, a'_n, \pi_n)) & \text{if } \bigwedge_{1 \le j \le h} a_j = a'_j \\ (a'_1, \ldots, a'_n) & \text{otherwise} \end{cases}$$

6. the stop operator returns true only if all analysis components agree:

$$\mathsf{stop}_{\mathbb{D}}((a_1,\ldots,a_n),\mathsf{reached},(\pi_1,\ldots,\pi_n)) = \bigwedge_{1\leq i\leq n} \mathsf{stop}_{\mathbb{D}}(a_i,\mathsf{sreached}_i,\pi_i)$$

where $\mathsf{sreached}_i$ is the set of the ith components of those reached states where the h state splitting components are equal to those in $(a_1,\ldots,a_n)$:

$$\mathsf{sreached}_i = \{r_i \mid h < i \leq n \wedge (a_0,\ldots,a_h,r_{h+1},\ldots,r_i,\ldots r_n) \in \mathsf{reached}\}$$

7. the componentwise precision adjustment operator is

$$\mathsf{prec}_{\mathbb{D}}((a_1,\ldots,a_n),(\pi_1,\ldots,\pi_n),\mathsf{reached}) = (a'_1,\ldots,a'_n),(\pi'_1,\ldots,\pi'_n)$$

with $\mathsf{prec}_{\mathbb{A}_i}(a_i,\pi_i,\mathsf{sreached}_i) = (a'_i,\pi'_i)$ for all $1 \leq i \leq n$ using the set $\mathsf{sreached}_i$ defined above.

This generic composite analysis lays a solid foundation for easily combining multiple analyses. It was successfully used in all experiments and provided sufficient precision for the executables and properties considered. For more fine tuning of the interplay between analyses, however, the individual operators, in particular the composite precision adjustment operator, can be fully customized in a derived composite domain (implementing the custom composition as a subclass, for example).

5.3 Abstract Domains in Jakstab

Several abstract domains have been formalized as CPAs and implemented for use with Jakstab. The CPA framework allows to combine analyses freely, and in principle, the algorithm for control flow reconstruction will run with any combination of abstract domains. However, as discussed in Section 4.1, there are some requirements on the combined abstract domain for achieving a reasonable overapproximation of the control flow graph in presence of indirect control flow. Even in programs that do not contain any indirect jumps or calls, return addresses need to be represented. Consequently, any program containing procedures will need to be processed using either an explicit value domain or a call stack analysis, in which each call site is stored by its concrete address.

5.3.1 Location Analysis

Formulating a forward location analysis allows to factor out the notion of program locations from other analyses. A location analysis simply records the current program counter location as its abstract state. If a composite analysis is constructed as the direct product of location analysis with some other CPA, the composite abstract state contains location information that can (and usually will) be used by the composite merge operator to only merge states at the same location. The location CPA is, for the most part, directly taken from [17]. It is defined as $(\mathcal{L}_{\mathbf{L}}, \gamma_{\mathbb{L}}, \Pi_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, \mathsf{merge}_{\mathbb{L}}, \mathsf{stop}_{\mathbb{L}}, \mathsf{prec}_{\mathbb{L}})$, where

1. $\mathcal{L}_{\mathbf{L}} = \langle \mathbf{L} \cup \{\top_{\mathbf{L}}, \bot_{\mathbf{L}}\}, \top_{\mathbf{L}}, \bot_{\mathbf{L}}, \sqsubseteq_{\mathbf{L}}, \sqcup_{\mathbf{L}} \rangle$ is the flat lattice of incomparable locations (i.e., $\forall \ell, \ell' \in \mathbf{L}.\, \ell \sqsubseteq_{\mathbf{L}} \ell' \implies \ell = \ell'$) extended with a top element $\top_{\mathbf{L}}$ representing all locations and a bottom element $\bot_{\mathbf{L}}$ representing no location, such that $\forall \ell \in \mathbf{L}.\, \bot_{\mathbf{L}} \sqsubseteq_{\mathbf{L}} \ell \sqsubseteq_{\mathbf{L}} \top_{\mathbf{L}}$.

2. the concretization function maps location elements to the states where the program counter holds the respective address value:

$$\gamma_{\mathbb{L}}(\ell) = \begin{cases} \{s \mid s \in S, s(pc) = \ell\} & \text{if } \ell \neq \top_{\mathbf{L}} \text{ and } \ell \neq \bot_{\mathbf{L}} \\ S & \text{if } \ell = \top_{\mathbf{L}} \\ \emptyset & \text{if } \bot_{\mathbf{L}} \end{cases}$$

3. the set of precisions $\Pi_{\mathbb{L}} = \{null\}$ is the singleton set of *null*, as location analysis does not use precision adjustment.

4. $\rightsquigarrow_{\mathbb{L}}$ is the transfer relation relating locations that are connected by edges with $\forall \ell, \ell' \in \mathbf{L}.\, \ell \overset{g}{\rightsquigarrow}_{\mathbb{L}} \ell' \Leftrightarrow g = (\ell, \cdot, \ell') \vee \ell = \ell' = \top$.

A corresponding backward location analysis can be constructed by reversing the transfer relation such that $\forall \ell, \ell' \in \mathbf{L}.\, \ell' \overset{g}{\rightsquigarrow}_{\mathbb{L}} \ell \Leftrightarrow g = (\ell, \cdot, \ell') \vee \ell = \ell' = \top$. Note that a backward location analysis requires the use of a backward transformer factory and cannot be used for resolving control flow.

5. $\mathsf{merge}_{\mathbb{L}}(\ell, \ell', \pi) = \ell'$ keeps location elements separate.

6. $\mathsf{stop}_{\mathbb{L}}(\ell, \mathsf{reached}, \pi) = \ell \in \mathsf{reached}$ checks whether a location has already been reached.

7. the precision is unused, i.e., $\mathsf{prec}_{\mathbb{L}}(\ell, \pi, \mathsf{reached}) = (\ell, \pi)$.

5.3.2 Bounded Address Tracking

Bounded Address Tracking is used as the default abstract domain in Jakstab, offering a very precise path sensitive analysis for small sized, untrusted binaries. The concept is discussed in depth in Chapter 4, now we will define the operators to fit Bounded Address Tracking within the CPA framework. This definition also allows to set individual bounds for specific registers and memory regions and closely matches the actual implementation within Jakstab. The mapping to CPA requires some adaptation of the abstract $\widehat{\mathbf{post}}$ operator and the BOUND rule, whose functionality is divided among the transfer relation and the precision adjustment operator. The CPA for Bounded Address Tracking is defined as $\mathbb{B} = (\mathcal{L}_{\mathbb{B}}, \gamma_{\mathbb{B}}, \Pi_{\mathbb{B}}, \rightsquigarrow_{\mathbb{B}}, \mathsf{merge}_{\mathbb{B}}, \mathsf{stop}_{\mathbb{B}}, \mathsf{prec}_{\mathbb{B}})$, where

1. $\mathcal{L}_{\mathbb{B}} = \langle (V \cup \hat{A}) \times \hat{A}, \top_{\mathbb{B}}, \bot_{\mathbb{B}}, \sqsubseteq_{\mathbb{B}}, \sqcup_{\mathbb{B}} \rangle$ is the semilattice of abstract address valuations, i.e., a product lattice of the individual mappings from registers and memory locations to abstract addresses in $\hat{A}$.

2. $\gamma_{\mathbb{B}}$ is the concretization function that maps abstract states to sets of concrete states by enumerating all combinations of possible base addresses for abstract memory regions and all possible concrete values for unknown offsets or regions.

3. the set of precisions $\Pi_{\mathbb{B}} = (V \cup \hat{A}) \rightarrow (\mathbb{N}_0 \times \mathbb{N}_0)$ that associate to each register and memory location a pair of thresholds for the number of memory regions and the number of values to track. For instance, the precision $\{\texttt{eax} \mapsto (5,0), (\mathsf{stack}, -4) \mapsto (5,10)\}$ specifies that the analysis should precisely track up to 5 different regions but no offsets for the `eax` register, and that for the memory location at offset -4 on the stack it should track up to 5 regions and 10 different offsets.

4. the transfer relation $\rightsquigarrow_{\mathbb{B}}$ is derived from the abstract post operator defined in Section 4.4. The use of the Bound rule is encoded into $\rightsquigarrow_{\mathbb{B}}$ using the precision π:

$$
\begin{aligned}
a \overset{(\ell, stmt, \ell')}{\rightsquigarrow_{\mathbb{B}}} (a'', \pi) :\Longleftrightarrow{}& \exists a' \in \hat{A}.\, a' = \widehat{\mathbf{post}}[\![stmt_{\ell'}^{\ell}]\!](a) \wedge a''(pc) = a'(pc) \\
&\wedge \forall v_b \in V.\, \mathsf{let}(r, o_b) = a'(v_b), \\
&a''(v_b) = \begin{cases} (\top_R, \top_b) & \text{if } \pi(v_b) = (0,0) \\ (r, \top_b) & \text{if } \pi(v_b) = (k_r, 0), k_r \in \mathbb{N}_0, k_r > 0 \\ (r, o_b) & \text{otherwise} \end{cases} \\
&\wedge \forall (r', o') \in \hat{A}.\, \mathsf{let}(r, o_b) = a'(\hat{m}_b(r', o')), \\
&a''(r', o') = \begin{cases} (\top_R, \top_b) & \text{if } \pi(r', o') = (0,0) \\ (r, \top_b) & \text{if } \pi(r', o') = (k_r, 0), k_r \in \mathbb{N}_0, k_r > 0 \\ (r, o) & \text{otherwise} \end{cases}
\end{aligned}
$$

If the precision specifies a value bound of 0 for both region and offset of abstract values of a specific register $v \in V$ or memory location $(r, o) \in \hat{A}$ of bit length b, this register or memory location is set to $(\top_R, \top_b)$ in the abstract successor state a'' (first case in both case distinctions). If only the offset is set to a bound of 0, the region is kept and only the offset set to $\top_b$ (second case in both case distinction). If the bound is still set to some non-zero positive value, both region and offset are tracked at full precision.

Note that, in contrast to the semantics definition in Section 4.4, the transfer relation does not invoke an explicit bound operator for counting and widening values. Instead, the transfer relation uses the information stored in the precision element π. In the CPA model, the precision is updated only by the precision operator prec. The transfer relation then respects the precision information when calculating successor states.

5. the merge operator

$$\mathsf{merge}_{\mathbb{B}}(a, a', \pi) = \begin{cases} a & \text{if } a' \sqsubseteq_{\hat{A}} a \\ a' & \text{otherwise} \end{cases}$$

keeps states separate unless the new state is greater or equal. It is different from the default separating merge_{sep} operator introduced in [17] in that it joins an existing state a' with the new state a if a overapproximates a' (note that $a \sqsupseteq_{\hat{A}} a' \Rightarrow a \sqcup a' = a$). This check allows to remove states from the set of reached states if they are subsumed by a new state. The subsumption check helps to save memory, which can become critical when a high value bound is used.

6. the stop operator, defined as $\mathsf{stop}_{\mathbb{B}}(a, \mathsf{reached}, \pi) = \exists a' \in \mathsf{reached}. a \sqsupseteq a'$, checks whether the new state is already subsumed by a single existing state in the set of reached states.

7. the precision adjustment operator $\mathsf{prec}(a, \pi, \mathsf{reached})$ counts the number of distinct values and regions per register and memory location and checks whether that number exceeds the previous value or region bound stored in π. If it does, the new precision will contain a zero bound for the number of offsets or regions to track, respectively.

   Note that, to be equivalent to the formulation of Chapter 4, the analysis has to be combined with a location analysis such that the set $\mathsf{reached}$ passed to prec is limited to the states at a particular location.

The precision adjustment operator is formally defined as

$$\begin{aligned}\mathsf{prec}(a,\pi,\mathsf{reached}) :=\{(x,k_r,k_o) \mid & \exists x \in (V \cup \hat{A}), k_r', k_o' \in \mathbb{N}_0.\ (x,k_r',k_o') \in \pi,\\ & k_r = \begin{cases} k_r' & \text{if } \|\{r \mid s(x) = (r,o).s \in \mathsf{reached}\}\| \le k_r' \\ 0 & \text{otherwise}\end{cases},\\ & k_o = \begin{cases} k_o' & \text{if } \|\{s(x) \mid s \in \mathsf{reached}\}\| \le k_o' \\ 0 & \text{otherwise}\end{cases}\\ &\}\end{aligned}$$

5.3.3 Constant Propagation

To implement fast and simple disassembly with high coverage in the style of IDA Pro, Jakstab supports a coarse constant propagation (including constant folding) to resolve indirect function calls where the compiler stores constant addresses in intermediate registers for speed [81]. It is formulated as the CPA $\mathbb{C} = (\mathcal{L}_\mathbb{C}, \gamma_\mathbb{C}, \Pi_\mathbb{C}, \rightsquigarrow_\mathbb{C}, \mathsf{merge}_\mathbb{C}, \mathsf{stop}_\mathbb{C}, \mathsf{prec}_\mathbb{C})$ where

1. $\mathcal{L}_\mathbb{C} = \langle (V \to \mathcal{L}_{\mathbb{I}_*}) \cup \{\top_\mathbb{C}, \bot_\mathbb{C}\}, \top_\mathbb{C}, \bot_\mathbb{C}, \sqsubseteq_\mathbb{C}, \sqcup_\mathbb{C}\rangle$ is the lattice of mappings of registers into the flat lattice $\mathcal{L}_{\mathbb{I}_*} = (\mathbb{I}_* \cup \{\top_{\mathbb{I}_*}, \bot_{\mathbb{I}_*}\}, \top_{\mathbb{I}_*}, \bot_{\mathbb{I}_*}, \sqsubseteq_{\mathbb{I}_*}, \sqcup_{\mathbb{I}_*})$ of incomparable bit-vectors (i.e., $\forall z, z' \in \mathbb{I}_*.\ z \sqsubseteq_{\mathbb{I}_*} z' \implies z = z'$ and $\forall z \in \mathbb{I}_*.\ \bot_{\mathbb{I}_*} \sqsubseteq_{\mathbb{I}_*} z \sqsubseteq_{\mathbb{I}_*} \top_{\mathbb{I}_*}$). The bottom element $\bot_\mathbb{C}$ denotes that a state has not been reached by the analysis and corresponds to the empty mapping, the top element $\top_\mathbb{C}$, which is also the initial element for the analysis, maps all registers to the unknown value $\top_{\mathbb{I}_*}$. The partial order $\sqsubseteq_\mathbb{C}$ is defined as the pointwise ordering of the individual mappings of $V \to \mathcal{L}_{\mathbb{I}_*}$, and $\sqcup_\mathbb{C}$ is accordingly defined such that different register mappings are joined to $\top_{\mathbb{I}_*}$.

2. the concretization function $\gamma_\mathbb{C}$ maps constant values to their value, and unknown values to all possible combinations of values:

$$\gamma_{\mathbb{C}}(a) = \begin{cases} \{s \mid s \in S \wedge \forall v \in V.\, a(v) \neq \top_{\mathbb{I}_*} \implies s(v) = a(v)\} & \text{if } a \neq \bot_{\mathbb{C}} \\ \emptyset & \text{if } a = \bot_{\mathbb{C}} \end{cases}$$

3. the set of precisions $\Pi_{\mathbb{C}} = \{\mathit{null}\}$ contains only the single *null* element.

4. the transfer relation is defined by

   $$a \overset{g}{\rightsquigarrow}_{\mathbb{C}} a' :\Longleftrightarrow$$
   $$a' = \begin{cases} a[v \mapsto \widehat{\mathbf{eval}}[\![e]\!](a)] & \text{if } g = (\cdot, v := e, \cdot), v \in V, e \in \mathbf{Exp} \\ a[v \mapsto \top_{\mathbb{I}_*}] & \text{if } g = (\cdot, \mathit{stmt}, \cdot), \mathit{stmt} \in \{\mathsf{alloc}\ v, \cdot, \mathsf{havoc}\ v <_u \cdot\} \\ a & \text{otherwise} \end{cases}$$

   where $\widehat{\mathbf{eval}}$ is an abstract evaluation operator similar to the one defined for Bounded Address Tracking defined in Section 4.4. In fact, the same definition can be used if all constant values are paired with the global memory region.

5. the operator $\mathsf{merge}_{\mathbb{C}}(a, a', \pi) = a \sqcup_{\mathbb{C}} a'$ joins states. When combined with a standard location analysis that splits states by location, this yields the classic constant propagation that identifies for each location those registers that are guaranteed to have some constant value in all possible executions. Without a location analysis, this will find those registers which are *globally* constant.

6. $\mathsf{stop}_{\mathbb{C}}(a, \mathsf{reached}, \pi) = \exists a' \in \mathsf{reached}.\, a \sqsubseteq_{\mathbb{C}} a'$ checks whether the current constant valuation is subsumed by a state that has already been reached, i.e., a state where strictly less registers are constant.

7. the precision is never adjusted, i.e., $\mathsf{prec}_{\mathbb{C}}(a, \pi, \mathsf{reached}) = (a, \pi)$.

Constant propagation on its own is not expressive enough for control flow reconstruction as it does not maintain a model of the stack. Therefore it has to be combined with a call stack analysis to allow control flow reconstruction from programs with more than a single procedure. It is most effectively used in a fast and unsound disassembly run together with the optimistic transformer factory which does not require a call stack.

5.3.4 Strided Interval Analysis

Strided interval analysis is a path insensitive data flow analysis using intervals with congruence information, i.e., only every nth element is included in an interval of stride n. It is especially useful for dealing with switch statements for which the value being switched on is unknown, e.g., because it depends on input that can take a wide range of values.

Switch Statements. For example, a C switch statement with the switch expression x, case values 1 and 2, and a default case, is of the form:

```
switch(x) { // switch expression x
    case 1:  // case value 1
        break;
    case 3:  // case value 3
        break;
    default: // default case
}
```

Depending on the distribution of case values, an optimizing compiler translates high level switch statements into indirect jumps that read the jump target from a table of address values, a *jump table*, which is stored in the static data region of the executable. The compiler inserts computations before the jump to convert the value of switch expressions to indices in the jump table. This arithmetic code can show some variation and can even include a second, larger table of byte values, which define indices in the primary jump table. In general, however, switch statements are compiled according to the following pattern:

$$\begin{aligned}&\text{if } x >_u max \text{ jmp default}\\&x := x - min\\&\text{if } 1_1 \text{ jmp } m_{32}[\text{table\_base} + x]\end{aligned}$$

Here, x is the switch expression, *min* the smallest and *max* the largest unsigned case value. The jump table is located at address table_base and extends until table_base $+ \, max - min$. If x is determined by some input which is not explicitly abstracted by havoc in the execution environment, or if the number of potential values is larger than the value bound, Bounded Address Tracking cannot track concrete values for x and thus fail to resolve the targets of the indirect jump. An interval analysis, however, can deduce information from the first guarded jump, which will be resolved to assume $x >_u max$, and establish that $x <_u max$. After the subtraction, it will determine that $0 <_u x <_u (max - min)$. Thus the target expression of the indirect jump is correctly determined to be within the range of the jump table and the strided interval analysis can provide information about the entries to the control flow reconstruction.

Recall that **resolve** invokes the abstract expression evaluation $\widehat{\mathbf{eval}}$ to retrieve the set of concrete addresses that the target expression can represent in the current abstract state (see Section 3.4.1). In the implementation of $\widehat{\mathbf{eval}}$, it is important that the data of all table entries in the interval is returned as a set of concrete addresses, without an intermediate abstraction step to the interval domain. Otherwise, information is lost and spurious jump targets can be introduced.

Strided Interval CPA. The implementation of interval analysis in Jakstab is geared towards assisting in dealing with jump tables and is not meant to be executed as a standalone analysis. It is formalized as the CPA $\mathbb{T} = (\mathcal{L}_{\mathbb{T}}, \gamma_{\mathbb{T}}, \Pi_{\mathbb{T}}, \rightsquigarrow_{\mathbb{T}}, \mathsf{merge}_{\mathbb{T}}, \mathsf{stop}_{\mathbb{T}}, \mathsf{prec}_{\mathbb{T}})$, where

1. $\mathcal{L}_{\mathbb{T}} = \langle (V \cup \hat{A}) \to I, \top_{\mathbb{T}}, \bot_{\mathbb{T}}, \sqsubseteq_{\mathbb{T}}, \sqcup_{\mathbb{T}} \rangle$ is the product lattice of the individual mappings of registers and memory locations to the set I of *reduced strided intervals*. Strided intervals are written $q[x; y]$ and represent all num-

bers $x + q \cdot i$ with $0 < i < \frac{y-x}{q}$. Since the intervals are reduced, $y - x$ is always a multiple of q.

2. the concretization function $\gamma_{\mathbb{T}}$ is defined as the product of the concretization functions for intervals, which maps an interval to the set of all its contained elements.

3. the unused set of precisions $\Pi_{\mathbb{T}} = \{null\}$.

4. the transfer relation implements arithmetic over strided intervals according to [5]. To illustrate the example above, the transfer relation for assume edges of the form $g = (\cdot, \mathsf{assume}\ x \leq_u n, \cdot$ is $a \overset{g}{\rightsquigarrow} a[x \mapsto 1[0; n]$, i.e., the state is updated with the information that variable x lies in the interval between 0 and the constant n, with a stride of 1.

5. the merge operator implements the simple strategy of widening whenever control flow joins. It is defined as the product of the widening operator $\nabla :: I \times I \rightarrow I$ applied on the individual strided intervals in the variable valuation:

$$\begin{aligned}\mathsf{merge}_{\mathbb{T}}(a, a', \pi) = a'' :\Longleftrightarrow & \forall v \in V.\, a''(v) = a'(v) \nabla a(v) \wedge \\ & \forall (r, o) \in \hat{A}.\, a''(r, o) = a'(r, o) \nabla a(r, o)\end{aligned}$$

   For a definition of ∇ for reduced strided intervals, refer to [5]. A more sophisticated widening strategy would require reasoning about the structure of the control flow graph, but is not necessary for the purpose of resolving switch jumps.

6. $\mathsf{stop}_{\mathbb{T}}(a, \mathsf{reached}, \pi) = a \sqsubseteq_{\mathbb{T}} \bigsqcup \mathsf{reached}$ checks whether the current state is subsumed by the join of all reached states, i.e., whether greater or equal intervals are known for all variables. Note that reached contains at most one state in the default configuration, where states are immediately merged.

7. $\mathsf{prec}_{\mathbb{T}}(a, \pi, \mathsf{reached}) = (a, \pi)$ does not adjust the precision.

5.3.5 Call Stack Analysis

Call stack analysis, or call strings, are a common method for achieving context sensitive interprocedural analysis in source languages. In Jakstab, call stack analysis further serves as a method for tracing interprocedural control flow. Since the general CPA algorithm essentially inlines procedures, maintaining a call stack (or a model of the full, actual stack) is necessary to determine the targets of return instructions.

For binaries, a call stack approach is generally not sound, as the concept of procedures is not enforced and procedure calls and returns can be freely used for arbitrary jumps. Nevertheless, for binary analysis use cases where the compiler and executable are trusted, well-behavedness of procedure calls and returns can be assumed and a call stack analysis offers a less expensive alternative to the explicit and path sensitive modeling of stack memory. Implementing the analysis requires an extension to the IL definition in Chapter 2: to differentiate calls and returns, CFA edges are annotated with the additional information whether the edge is derived from a call or a return instruction. Jakstab supports a call stack analysis that differentiates call sites by their return address and truncates recursive function calls from the same location, i.e., merges recursive invocations in the same state. It is formalized as the CPA $\mathbb{S} = (\mathcal{L}_\mathbb{S}, \gamma_\mathbb{S}, \Pi_\mathbb{S}, \rightsquigarrow_\mathbb{S}, \mathsf{merge}_\mathbb{S}, \mathsf{stop}_\mathbb{S}, \mathsf{prec}_\mathbb{S})$, where

1. $\mathcal{L}_\mathbb{S} = \langle \mathbf{U}, \top_\mathbb{S}, \bot_\mathbb{S}, \sqsubseteq_\mathbb{S}, \sqcup_\mathbb{S} \rangle$ is the flat lattice of incomparable sequences of unique locations. The call stack analysis truncates recursion, i.e., return locations appear at most once in each call stack. Since the set of addresses is finite, the set $\mathbf{U}$ of unique location sequences is finite and every element $a \in \mathbf{U}$ is finite.

2. the concretization function $\gamma_\mathbb{S}$ maps each abstract call stack $a = c_0 c_1 \ldots c_n$ to those concrete states from S whose sequence of return values stored on the stack is $r_0 r_1 \ldots r_n$ with $r_i = c_i d_0 d_1 \ldots d_{m_i} c_i$; i.e., each return address

in the abstract call stack can represent a collapsed recursive invocation sequence via intermediate procedure calls $d_0 \dots d_{m_i}$.

A transformer factory can use the call stack information when concretizing the value of a return address expression.

3. the unused set of precisions $\Pi_S = \{null\}$.

4. the transfer relation is defined as

$$\begin{array}{ll} c_0c_1\dots c_n \overset{call}{\rightsquigarrow}_S c_0c_1\dots c_n r & \text{for call edges, with } r \neq c_i, 0 \leq i \leq n \\ c_0c_1\dots c_n \overset{call}{\rightsquigarrow}_S c_0c_1\dots c_j & \text{for call edges, with } r = c_j \land r \neq c_i, 0 \leq i < j \\ c_0c_1\dots c_n \overset{ret}{\rightsquigarrow}_S c_0c_1\dots c_{n-1} & \text{for return edges} \\ c_0c_1\dots c_n \overset{g}{\rightsquigarrow}_S c_0c_1\dots c_n & \text{for other edges} \end{array}$$

where r denotes the address of the fall through statement of a call.

5. the operator $\mathsf{merge}_S(a, a', \pi) = a'$ keeps calling contexts separate.

6. $\mathsf{stop}_S(a, \mathsf{reached}, \pi) = a \in \mathsf{reached}$ checks whether the current calling context has already been reached.

7. $\mathsf{prec}_S(a, \pi, \mathsf{reached}) = (a, \pi)$ does not adjust the precision.

5.3.6 Forward Expression Substitution

The x86 assembly idioms for conditional statements, i.e., a comparison instruction and a subsequent conditional jump, decouples the condition from the location of the branch. The result of the comparison is stored in the EFLAGS register as a combination of arithmetic flags. This makes it harder for analyses to relate arithmetic conditions to conditional branches, essentially requiring a relational approach. The usefulness of strided interval domain, for example, depends on the kind of conditions that are assumed. While it cannot deduce much from assuming a Boolean combination of flags to be true, it can directly deduce value ranges from statements such as `assume` $eax < 10$.

An elegant way of dealing with this challenge is to use a forward expression substitution that rebuilds a high-level branch condition from the combination of status flags. The substituted expressions can then be used by other, non-relational analyses, allowing them to relate registers directly to branch conditions. For instance, consider the sequence of instructions `cmp eax, ebx; jl label` that executes a conditional branch if `eax` is less than `ebx`, and translates to this sequence of IL statements:

$$\begin{aligned}
&CF := (eax <_u ebx)\\
&OF := (eax < 0 \wedge ebx \geq 0 \wedge eax - ebx > 0) \vee\\
&\qquad\qquad (eax \geq 0 \wedge ebx < 0 \wedge eax - ebx < 0)\\
&SF := (eax - ebx < 0)\\
&ZF := (eax = ebx)\\
&\texttt{if } (SF \veebar OF) \texttt{ jmp label}
\end{aligned}$$

A forward expression substitution will determine the expression values of flags in the final guarded jump, and substitute the jump condition $(SF \veebar OF)$ with the expression

$$\begin{aligned}
&(eax - ebx < 0) \veebar\\
&\quad ((eax < 0 \wedge ebx \geq 0 \wedge eax - ebx > 0) \vee (eax \geq 0 \wedge ebx < 0 \wedge eax - ebx < 0)).
\end{aligned}$$

This can be simplified to

$$(eax < 0 \wedge ebx \geq 0) \vee (eax < 0 \wedge eax - ebx < 0) \vee (ebx \geq 0 \wedge eax - ebx < 0),$$

which in bit-vector arithmetic is equivalent to $(eax < ebx)$. Note that all subtraction refers to bit-vectors, i.e., uses modulo arithmetic. Since there are only a limited number of different patterns that arise from combinations of flags, a template based approach can quickly simplify these kinds of expressions.

The CPA for forward expression substitution, which implements this analysis, is defined as $\mathbb{E} = (\mathcal{L}_\mathbb{E}, \gamma_\mathbb{E}, \Pi_\mathbb{E}, \rightsquigarrow_\mathbb{E}, \mathsf{merge}_\mathbb{E}, \mathsf{stop}_\mathbb{E}, \mathsf{prec}_\mathbb{E})$, where

1. $\mathcal{L}_\mathbb{E} = \langle V \to \mathcal{L}_{\mathbf{Exp}}, \top_\mathbb{E}, \bot_\mathbb{E}, \sqsubseteq_\mathbb{E}, \sqcup_\mathbb{E} \rangle$ is the lattice of mappings of registers into the flat lattice $\mathcal{L}_{\mathbf{Exp}} = (\mathbf{Exp} \cup \{\top_{\mathbf{Exp}}, \bot_{\mathbf{Exp}}\}, \top_{\mathbf{Exp}}, \bot_{\mathbf{Exp}}, \sqsubseteq_{\mathbf{Exp}}, \sqcup_{\mathbf{Exp}})$ of incomparable symbolic expressions (i.e., $\forall e, e' \in \mathbf{Exp}.\, e \sqsubseteq_{\mathbf{Exp}} e' \implies e = e'$ and $\forall e \in \mathbf{Exp}.\, \bot_{\mathbf{Exp}} \sqsubseteq_{\mathbf{Exp}} e \sqsubseteq_{\mathbf{Exp}} \top_{\mathbf{Exp}}$). The bottom element $\bot_\mathbb{E}$ corresponds to the empty mapping, and the top element $\top_\mathbb{E}$ maps all registers to the unknown expression $\top_{\mathbf{Exp}}$. The partial order $\sqsubseteq_\mathbb{E}$ is then defined as the pointwise ordering of the individual mappings of $V \to \mathcal{L}_{\mathbf{Exp}}$, and $\sqcup_\mathbb{E}$ is defined accordingly.

2. $\gamma_\mathbb{E}$ is the concretization function, which does not follow the CPA standard as determining symbolic expressions for registers is not covered by the forward reachability collecting semantics. The collecting semantics for forward expression substitution records for each register all symbolic expressions that are equivalent to the registers contents, and each concrete element is a valuation $V \to 2^{\mathcal{L}_{\mathbf{Exp}}}$. Thus, $\gamma_\mathbb{E}(a) :: (V \to \mathcal{L}_{\mathbf{Exp}}) \to (V \to 2^{\mathcal{L}_{\mathbf{Exp}}})$ maps the abstract symbolic expression valuation a to all those concrete elements where the abstract expression substitutions are included in the set of equivalent expressions.

3. the set of precisions is unused and $\Pi_\mathbb{E} = \{null\}$.

4. the transfer relation stores after an assignment the fact that the left-hand side is equal to the right-hand side, and invalidates all previously stored facts which involve the left-hand side or an expression possibly aliasing it.

5. the operator $\mathsf{merge}_\mathbb{E}(a, a', \pi) = a \sqcup_\mathbb{E} a'$ joins states to allow the analysis to compute those expressions which are equivalent to a register on all paths to the current location (if the analysis is combined with a location analysis).

6. $\mathsf{stop}_{\mathbb{E}}(a, \mathsf{reached}, \pi) = \exists a' \in \mathsf{reached}.\, a \sqsubseteq_{\mathbb{E}} a'$ checks whether the current abstract state is subsumed by a state that has already been reached.

7. the precision remains constant, i.e., $\mathsf{prec}_{\mathbb{E}}(a, \pi, \mathsf{reached}) = (a, \pi)$.

Other analyses that want to use the substituted expressions can in principle implement a strengthening operator $\downarrow$ to improve their calculated abstract states. This is inefficient and inconvenient in practice, however; a more succinct solution is to slightly modify the composite transfer relation such that it creates new CFA edges with substituted expressions and passes these to the other analysis components. This way, other analyses require no modification to use the results of forward expression substitution.

5.3.7 Live Variable Analysis

One of the classical, simple program analyses is *Live Variable Analysis*, which determines for each program point which variables are *live*, i.e., whose contents may be used at a later time in the execution. It is a backward analysis, since it propagates information against the direction of control flow. Therefore it can only operate on a fully reconstructed CFA and has to be executed as a secondary analysis using the reverse transformer factory. It is defined as the CPA $\mathbb{V} = (\mathcal{L}_{\mathbf{V}}, \gamma_{\mathbb{V}}, \Pi_{\mathbb{V}}, \rightsquigarrow_{\mathbb{V}}, \mathsf{merge}_{\mathbb{V}}, \mathsf{stop}_{\mathbb{V}}, \mathsf{prec}_{\mathbb{V}})$, where

1. $\mathcal{L}_{\mathbf{V}} = \langle 2^V, V, \emptyset, \subseteq, \cup \rangle$ is the power set lattice for V ordered by subset inclusion, where each element denotes the set of registers live in the current state.

2. the concretization function $\gamma_{\mathbb{V}} :: 2^V \to 2^{2^V}$ does not follow the CPA standard but maps into sets of registers live on backward execution traces. The concrete collecting semantics for live variables is a backward semantics that records the set of registers that are live, i.e., actually used in the future. The abstract semantics of live variable analysis overapproximates the concrete set of live registers with a possibly larger set. Therefore the

concretization function is defined as $\gamma_{\mathbb{V}}(a) = 2^a$, which expresses that all subsets of the abstract set of live registers are possible concrete states.

3. the set of precisions $\Pi_{\mathbb{V}} = \{null\}$ is not used.
4. the transfer relation $\rightsquigarrow_{\mathbb{V}}$ is defined as $a \overset{g}{\rightsquigarrow}_{\mathbb{V}} (a \cap kill(g)) \cup gen(g)$, where $kill(g)$ and $gen(g)$ denote the sets of registers overwritten and read by the statement of g, respectively.
5. the operator $\mathsf{merge}_{\mathbb{V}}(a, a', \pi) = a \cup a'$ joins the live sets of multiple successor statements.
6. $\mathsf{stop}_{\mathbb{V}}(a, \mathsf{reached}, \pi) = a \subseteq \bigcup \mathsf{reached}$ checks whether the newly determined set of live registers was already known to be live. Note that reached never contains more than one state, as new states are always merged.
7. $\mathsf{prec}_{\mathbb{V}}(a, \pi, \mathsf{reached}) = (a, \pi)$ does not adjust the precision.

If the live variable CPA is not composed with a location analysis for selective merging, it will only maintain a single global state and calculate the set of all registers which are live at some location in the entire program. Combined with a backward location analysis into a composite CPA that performs selective merging on equal locations, the CPA behaves like a classical live variable analysis.

5.4 Code Transformations

Mapping every assembly instruction to its semantic specification creates a program representation with obvious pieces of dead code. In particular, most of the status flags are not used but simply overwritten by later instructions. If a secondary is performed after the initial control flow reconstruction phase, the CFA can be simplified significantly in between by performing live variable analysis and dead code elimination. Forward substitution of expressions further reduces the amount of temporary variables used, allows to eliminate flag assignments, and leads to more natural conditional expressions.

Forward Expression Substitution. Besides using the results of the forward expression substitution CPA $\mathbb{E}$ online during analysis, it is also possible to rewrite the CFA to a semantically equivalent CFA. The forward expression substitution transformation determines the set of reachable states of $\mathbb{E}$ on the reconstructed CFA. Afterwards, it iterates over all reached program states (one per reachable location) and substitutes each register or memory location used by the statement in the outgoing CFA edge of the states' location that has an expression value not equal to $\top_{\mathbb{E}}$. As the substituted registers were determined to be equal to the expression on all paths by the analysis, the resulting program is equivalent.

Consider the instructions sequence `cmp eax, ebx; jl label` discussed in the description of the forward expression substitution analysis (Section 5.3.6), where the jump expression simplified to $(eax < ebx)$. A substitution of the entire statement using the results of the analysis thus yields `if` $(eax < ebx)$ `jmp label` for the guarded jump, which encodes the branching condition directly between the original registers and can help an analysis to maintain their correlation. If the values of the status flags are not used by any other statements, they are no longer live and the corresponding assignments can be removed in a dead code elimination step.

Dead Code Elimination. Live variable analysis can determine the set of registers that will be used after being assigned. Most flag updates from the semantics specification of arithmetic instructions are never used for conditional tests, so they can be removed. As noted above, expression substitution can eliminate register uses and thus the liveness of many remaining flags and temporary variables. A dead code elimination can then remove those CFA edges containing updates to dead (non live) registers. Interleaving multiple rounds of live variable analysis and dead code elimination until no more edges can be removed can substantially reduce the CFA size (about 30% in early experiments [81]). In the above example, all flag updates would be removed (given that the flags are not used after the jump), and only the guarded jump would remain. Besides speed improvements for subsequent secondary analysis, dead code elimination also greatly improves the readability of control flow automata for reverse engineering or debugging.

5.5 Related Work

This chapter touched several different fields in its description of the analysis architecture. The following paragraphs discuss the relevant literature in these fields and, where appropriate, compare the related work to the implementation choices made in Jakstab.

Static Analysis of Binaries. In [10] and [118], Balakrishnan and Reps briefly describe the facilities of CodeSurfer/x86 for model checking a weighted pushdown system constructed from the call graph of the binary. They build on the existing infrastructure for C of the regular CodeSurfer platform and are able to answer reachability queries while precisely modeling interprocedural control flow including recursion. As the other CodeSurfer/x86 work, their approach relies on the assumption that procedure calls are well-behaved and that the disassembly was correctly performed.

A detailed report on the verification of API usage specifications is given in [8]. There, the authors present DDA/x86, an extension to CodeSurfer/x86 for verifying device driver binaries. Since the regular context sensitive, but path insensitive data flow analysis proved to be unsuitable for verifying relevant properties, they extended their tool to incorporate the ESP approach [49], which adds limited path sensitivity to a data flow analysis. A detailed experimental comparison of DDA/x86 with Jakstab and a discussion of the DDA/x86's expressiveness in practice is given in Section 6.1.

In their recent combined static and dynamic analysis framework McVeto [132], Thakur et al. present an adaption of the DASH algorithm [15] to the analysis of binaries. They disassemble instructions along the trace of the observed program execution, which allows them to deal with overlapping instructions as long as they were executed in the trace. The analysis is property-driven and attempts to reach a certain program location by manipulating program inputs. A major problem for generalizing this approach to non-academic examples is that program inputs are not always clearly defined. Particularly in executables, inputs can come from a plethora of sources such as files, network, the system clock, etc., and in practice the set of controlled inputs has to be constrained [60, 102]. These practical constraints prevent an application of this approach to full disassembly of realistic programs, which would require generating a set of test inputs that explores all instructions (full statement coverage).

Summarization of Library Calls. Gopan and Reps [62] exemplify a technique for summarizing library calls in executable analysis. Using a numeric analysis, they synthesize summary transformers for safety properties for the library functions memset and lseek. These kind of summaries could serve as a more precise alternative to using default transformers for imported but unknown library functions in Jakstab's analysis. A method for generating precise function summaries from source code is described in [138]. It has to be investigated how this summarization method can be applied to untyped machine code.

Checking Plain Text Assembly. Maus et al. [94] describe an approach to the analysis of assembly code in plain text, which is linked together with high level modules. They syntactically translate assembly instructions to C, such that the instructions invoke predefined macros and operate on variables in the C program. The resulting C program is then verified with existing infrastructure based on Boogie [14]. While such an approach offers a very precise solution for dealing with inline assembly, if the assembly semantics are fully encoded in C, it does not deal with the problems that arise in executable analysis – e.g., jumps are assumed to only go to predefined labels. Chaki and Ivers [24] basically follow the same approach of translating assembly source code into C, but they use a predicate abstraction based model checker to check specifications on the resulting C program.

Generating Provably Safe Machine Code. A completely different approach for ensuring the safety of machine code is to add annotations to assembly and executables that are easy to verify for an analyzer before executing the program. Most notably, in proof carrying code [106] the compiler equips an executable with a formal proof of given safety rules. The proof can then be validated by an operating system at load time before it decides to execute the executable. Another approach to safe machine code is typed assembly language [101]. Here, not complete proofs are shipped together with compiled executable code, but instead the instructions are annotated with types and invariants. The annotations take less space than full proofs, but allow the operating system to regenerate the safety proof on its own.

Chapter 6

Experiments

This chapter presents experimental results for analyzing binaries with Jakstab and compares them to existing state-of-the-art tools. The experiments are divided into two parts: Precise analysis of relatively small driver binaries and heuristic disassembly and control flow reconstructions of general, large executables. Both sets of experiments analyze Windows binaries.

6.1 Analyzing Untrusted Driver Binaries

It remains to show that the proposed approach of integrating control flow reconstruction and precise static analysis in the form of Bounded Address tracking is usable in practice and advances the state of the art. In the experiments in this section, Jakstab is used to verify API specifications on several Windows device drivers. To allow a comparison with the existing approach by Balakrishnan and Reps [8], the same set of drivers from the Windows Windows Driver Development Kit is analyzed. The experiments show several advantages of Jakstab over their IDA Pro and CodeSurfer/x86 based driver analyzer DDA/x86. In particular, Jakstab

- analyzes the driver binaries without relying on a heuristics based external disassembler,
- avoids a false positive in one of the drivers,

- detects a potential memory safety violation,
- is directly applicable to compiled binaries without access to source code, so the experiments were extended to over 300 closed source drivers.

6.1.1 Motivation

Device drivers supplied by hardware manufacturers are a main source of bugs in modern operating systems. In Windows XP, device drivers have been reported to cause 85% of all crashes [130]. In parts, this can be attributed to the fact that most closed source drivers installed on consumer desktop systems today have never been exposed to formal analysis. Source code analysis tools like Microsoft's Static Driver Verifier (SDV) [12] are available for developers to statically check their software for conformance to specifications of the Windows driver API. The vendors, however, are not forced to use these analysis tools in development, and they are unwilling to submit their source code and intellectual property to an external analysis process. Certification programs such as the Windows Logo Program [99] thus have to rely on testing only, which cannot provide guarantees about all possible executions of a driver. Without vendor support, the only way to make these often hastily written, yet critical programs accessible to static analysis is to directly work at the binary level. If the analysis does not require source code or debug symbols, an analysis infrastructure can be created independently of active vendor support.

6.1.2 Windows Driver Model

The Windows Driver Model was introduced by Microsoft with Windows 98 and Windows 2000 as the new Windows device driver API [98, 110]. Compiled WDM device drivers use the file extension `.sys` instead of the usual `.exe`, but are otherwise standard PE executable files. They export only their entry point, which has to be the standardized DriverEntry routine. To load a driver, the system calls the driver's DriverEntry routine and passes it an allocated DriverObject structure to hold information about the driver itself, and a Unicode String RegistryPath that the driver can use to store persistent information in the Windows system registry. The DriverEntry routine is responsible for filling the DriverObject with pointers to the dispatch and device handling routines of the driver, which provide the actual functionality [98].

A modern device driver supporting Plug'n'Play (PnP), i.e., adding and removing devices without restarting the system, typically supports the AddDevice, RemoveDevice, and DriverUnload routines that are called by the system whenever a device is attached, a device is removed, or the driver should unload, respectively. Furthermore, the driver provides several dispatch methods for handling I/O requests. For every supported dispatch method, the driver stores its address in an array within the DriverObject, which is indexed by the driver API defined *major function code* of the requested I/O operation. Common dispatch routines handle reading from and writing to the device, or handling PnP requests. Parameters to the dispatch routines, which further specify the request are packed within an I/O request packet (IRP) structure.

The Windows Driver Model documentation [98] specifies, both implicitly and explicitly, a rich set of rules for its correct usage. For instance, if a driver calls the API method IoAcquireCancelSpinLock, it is required to call IoReleaseCancelSpinLock before calling IoAcquireCancelSpinLock again. SDV is shipped with a set of 66 rules for basic driver functionality, which the development team extracted from the WDM documentation and formalized as finite state machines [13].

6.1.3 OS Abstraction and Driver Harness

Executables in general and drivers in particular frequently interact with the operating system using calls to the system and driver API. As in source based analyses, these calls can be abstracted using stubs, which model the relevant side effects such as memory allocation or the invocation of callback routines. Following the approach of the source code software model checker SDV [12], Jakstab loads the driver along with a separate *harness* module, which includes system call abstractions relevant to drivers and contains a main function that non-deterministically exercises the driver's initialization and dispatch routines. The harness is written in C and compiled into an dynamic library (DLL) for loading; it is based on SDV's `osmodel.c` and follows SDV's invocation scheme for PnP drivers defined therein (using macros for invoking driver functions):

```
DriverEntry;
sdv_RunAddDevice;
sdv_RunStartDevice;
( DoNothing ||
  sdv_RunStartIo ||
  sdv_RunDPC ||
  sdv_RunISR ||
  sdv_RunCancelRoutine ||
  sdv_RunDispatchFunction );
sdv_RunRemoveDevice;
sdv_RunUnload;
```

That is, it first runs the DriverEntry routine (the main function of the driver executable) that initializes internal data structures and registers the other functions provided by the driver. From these the harness calls the functions for adding and starting a new physical device to allow the driver to set up all data for a new device. After these steps are complete, the harness nondeterministically executes either one of the registered dispatch functions for the various types of IRPs, or it invokes deferred procedure calls (DPCs), cancel routines, or interrupt service requests (ISR) set up by the driver. Finally, the harness signals to the driver that the device is removed and calls the driver's unload function.

For the experiments, the specifications were manually encoded into the harness by inserting state variables and assertions at the locations where SDV places hooks into its specification files. The IL statements alloc, free, havoc, and assert are exclusively generated by the harness, since they do not correspond to any real x86 instructions. These statements are encoded into the compiled harness object

IL Statement	ASM code	Notes
alloc v, e	`lock rep inc eax`	Implemented as function with allocated pointer as return value (`eax`).
free p	`lock rep not p`	`not` can take registers or memory operands.

assert $x >_u y$	`mov edx, x`
	`lock rep add edx, y`
		The second instruction is translated to the actual assertion.

assert $x \geq_u y$	`mov edx, x`
	`lock rep adc edx, y`	

assert $x = y$	`mov edx, x`
	`lock rep cmp edx, y`
		Currently only assertion expressions $>_u$, $\geq_u$, and $=$ are implemented.

havoc $v <_u x$	`lock rep sub eax, x`
	`mov v, eax`
		The first instruction is translated to the actual havoc statement, the second stores the result in the designated variable.

Table 6.1: (Pseudo-) instructions that can be inlined for using abstract IL statements in the C-language harness.

file using combinations of instructions and prefixes that are illegal according to official Intel documentation [74], but which are supported by the inline assembler nonetheless. During on-demand disassembly, Jakstab detects these instructions and directly maps them to the corresponding IL statements. The available instruction patterns to encode IL statements are listed in Table 6.1. Only the 32 bit variants of the instructions are shown; for each statement there are also 16 and 8 bit variants that use the 16 and 8 bit subregisters, respectively (e.g., `ax` and `al` for `eax`).

Several parts of the SDV harness and rules had to be modified to make it suitable for binary analysis. For example, the preprocessor macro IoMarkIrpPending, which sets a bit in the control word of IRPs, is intercepted by SDV to change the state for the *PendedCompletedRequest* rule. Since macro invocations are no longer explicit in the binary, the rule's assertion had to be modified to check the bit directly instead of a separate state variable. Furthermore, SDV's statement for non-determinism had to be replaced by either havoc or *nondet*, depending on the context.

Another issue arises from the fact that the SDV harness was not constructed with memory safety in mind. Many function stubs coarsely abstract return values where they actually should allocate memory, which leads to potential dereferences of uninitialized pointers. In several places it was therefore necessary to refine SDV's harness with explicit memory allocation. For the same reason it was also required to add rudimentary support for threads, which are used for initialization or packet handling by some drivers. The original SDV harness simply ignores calls to thread creating Windows API functions such as PsCreateSystemThread, and thus many data structures stay uninitialized. Jakstab does not model concurrency, but the harness can emulate the behavior of initialization threads that only run for a short amount of time by simply calling the thread's entry routine. In case of a long running thread (e.g., for IRP processing), this causes problems, however, when the thread loops forever until a shared flag is set from the main thread. In such cases the analysis will reach a fixpoint over the infinite loop and never exit the loop, leaving large parts of the real driver code unexplored. Without real support for reasoning about concurrency, the experiments used two different harness configurations depending on the nature of threads used in the drivers: Either calls to functions for thread creation are simply translated to calls of the thread's entry function, or they are skipped, which is the default behavior of the SDV harness. Which harness configuration was chosen for the experiments is shown in the column labeled **MT** of Table 6.3.

There are occasions where the execution model of the harness causes memory safety violations in the driver which are impossible at runtime. SDV's execution model for PnP drivers processes a deferred procedure call (DPC) if it has been registered by the driver. In actual executions, however, synchronization operations force the DPC to finish at some specific point in time. After that, the context of the procedure call (e.g., the IRP currently being processed) can become invalid. In this case, the harness invokes the DPC on an invalid context, which can cause false alarms on using uninitialized pointers. To alleviate this problem, a second configuration option for the Jakstab harness defines whether the objects representing IRPs are reinitialized between the non-deterministic method invocations. If IRPs are not reset but kept, the context stays valid for DPCs. A full solution to this issue would have to take synchronization and timer events into account, which are yet another concurrency mechanism besides threads.

6.1.4 Experimental Setup

For a direct comparison with the IDA Pro and CodeSurfer/x86 based binary driver analyzer DDA/x86 described in [8], Jakstab processed the same set of drivers from the Windows Driver Development Kit (DDK) release 3790.1830 and checked the same specification *PendedCompletedRequest*. The rule specifies that a driver must not call IoCompleteRequest and return STATUS_PENDING unless it invokes the IoMarkIrpPending macro on the IRP being processed. The DDK drivers were compiled without debug information and default settings. Note that in contrast to [8], the driver source code was not directly linked against the harness; this novel approach does not require special preparation and is directly applicable to drivers without access to source code.

Jakstab was configured to use Bounded Address Tracking as its only analysis. Through computing the set of reachable states over the harness and driver, Jakstab can check whether states are reachable that violate the assertions in the harness that encode the specifications to check. To eliminate unsoundness from the control flow analysis, the most precise pessimistic transformer factory was used for the experiments (see Section 5.2.3).

6.1.5 Results

The experimental results are listed alongside those reported in [8] in Table 6.2. The number of instructions include instructions from the harness in both cases.

	DDA/x86			Jakstab		
Driver	**Instr**	**Time**	**Result**	**Instr**	**Time**	**Result**
krnldrvr.sys	2824	14s	✓	413	2s	✓
ioctl/sys/sioctl.sys	3504	13s	✓	630	7s	✓
tracedrv.sys	3719	16s	✓	439	2s	✓
startio/cancel.sys	3861	12s	✓	759	2s	✓
sys/cancel.sys	4045	10s	✓	780	2s	✓
moufiltr.sys	4175	3m 3s	$\times$	722	9s	$\times$
event/sys/event.sys	4215	20s	✓	690	2s	✓
kbfiltr.sys	4228	2m 53s	$\times$	726	8s	$\times$
toastmon.sys	6261	4m 1s	✓	977	9s	✓
diskperf.sys	6584	3m 17s	✓	1409	46s	✓
fakemodem.sys	8747	11m 6s	✓	1887	24s	$\times_m$
flpydisk.sys	12752	1h 6m	FP	1782	39m34s	✓
mouclass.sys	13380	40m 26s	FP	1763	8s	FP_c
mouser/sermouse.sys	13989	1h 4m	FP	1293	4s	FP
SerEnum.sys	14123	19m 41s	✓	1503	8s	✓
1394DIAG.sys	23430	1h33m	FP	2426	4s	FP_m
1394VDEV.sys	23456	1h38m	FP	2872	5s	FP_m

Table 6.2: Comparison of experimental results on Windows DDK drivers between DDA/x86 (on a 3 GHz Xeon) and Jakstab (on a 3 GHz Opteron). Explanation of result types: ✓: specification verified, FP: infeasible counterexample, $\times$: feasible counterexample, $\times_m$: feasible trace to memory safety violation, FP_m: infeasible trace to memory safety violation, FP_c: infeasible trace to invalid control flow.

Driver	**k**	$\mathbf{k_h}$	**IRPs**	**MT**	**Stmts**	**BBs**	**Visited**	**Final**
krnldrvr.sys	28	5	keep	-	860	94	378	378
sioctl.sys	28	5	keep	-	1398	191	3947	587
tracedrv.sys	28	5	keep	-	956	105	486	371
startio/cancel.sys	28	5	keep	-	1787	187	633	614
sys/cancel.sys	28	5	keep	skip	1910	192	600	577
moufiltr.sys	28	5	reset	-	1544	178	3830	3740
event.sys	28	5	keep	-	1510	169	663	649
kbfiltr.sys	28	5	reset	-	1548	180	3834	3744
toastmon.sys	28	25	reset	-	1995	220	4853	4823
diskperf.sys	28	5	reset	-	3273	377	19772	18137
fakemodem.sys	28	5	keep	-	4236	613	13994	14044
flpydisk.sys	100	35	reset	call	4235	584	186543	130767
mouclass.sys	28	28	reset	-	4064	434	3055	3395
sermouse.sys	28	28	reset	-	3032	348	1888	2005
SerEnum.sys	28	25	reset	skip	3453	444	5213	4908
1394DIAG.sys	28	28	reset	-	5464	620	2181	2381
1394VDEV.sys	28	28	reset	-	6427	747	2837	2974

Table 6.3: Details of DDK experiments for Jakstab. **k**: value bound, $\mathbf{k_h}$: value bound for heap memory, **IRPs**: state reset for I/O request packets between calls, **MT**: handling of thread creation, if used, **Stmts**: Number of statements, **BBs:** Number of basic blocks, **Visited**: Number of visited states, **Final**: Size of the final state space after the algorithm finishes.

Note that the tools report very different numbers of instructions for the same binaries; this is due to the fact that Jakstab disassembles instructions only on demand, i.e., if they are reachable by the analysis. In contrast, CodeSurfer/x86 uses IDA Pro as front end, which heuristically disassembles all likely instructions in the executable. Since for DDA/x86 the entire harness was compiled and linked with the driver, IDA Pro disassembled all code from the harness, including harness code that is unreachable with the driver under analysis. Furthermore, it is possible that parts of the driver code are unreachable from the harness. The experiments used two value bounds which were determined empirically and are listed in Table 6.3: k shows the value bound for registers and stack locations, k_h the value bound for memory locations in allocated heap regions.

Performance. The comparison of execution times should be taken with a grain of salt, since both prototypes were run on different machines. DDA/x86 was run on a 64 bit Xeon 3 GHz processor with 4GB of memory per process, while the experiments with Jakstab were conducted on a 64 bit AMD Opteron 3 GHz processor with 4 GB of Java heap space (the average over 10 runs per driver is reported). Still, it is possible to see that execution times for Jakstab appear favorable overall.

Comparing Precision of Jakstab and DDA/x86. For `flpydisk.sys`, Jakstab was able to verify the specification, while DDA/x86 found a false positive (FP). DDA/x86 follows the ESP approach [49] for differentiating paths based on states of the property automaton. ESP can extend a classical data flow analysis that merges abstract states at join points in the control flow graph by simulating transitions of a property automaton during analysis. ESP keeps those states of the data flow analysis separate where the property automaton is in different states, and merges all those states which share the same state of the property automaton.

While ESP is an effective method to reduce the state space, it is difficult to find a property automaton that contains the right amount of abstraction. Since the property automaton derived from the *PendedCompletedRequest* rule caused too many false positives, the authors of DDA/x86 used an additional, hand-written automaton. It exploits the fact that most dispatch routines in drivers from the Windows DDK use a single variable for maintaining the error status before returning it to the caller. On every assignment to the local status variable, it changes state to `st_pending`, `st_not_pending`, or `st_unknown`, depending on whether the value assigned is STATUS_PENDING, a different value, or an unknown value. In `flpydisk.sys`, however, property relevant code is spread over multiple procedures defining their own status variables. DDA/x86 does implement an heuristic to identify status assignments in called procedures that affect the status variable [4], but this method can fail and lead to false positives. Figure 6.1 shows the relevant source code portions in `floppy.c` (for ease of explanation, source is preferred over assembly here) and illustrates the states reached by the combined VSA and ESP analysis in DDA/x86. The states are given in comments as triples of the value set for the status variable, the state of the status property automaton, and the state of the specification automaton as defined in [8].

Using VSA and ESP as in DDA/x86 on this code to check the *PendedCompletedRequest* rule yields the following false positive: The dispatch method FloppyReadWrite passes an incoming IRP to the auxiliary FlQueueIrpToThread method (line 21). At this time, the property automata are both still in the initial states `st_unknown` (variable ntStatus is uninitialized) and `start` (no rule-relevant actions have been observed so far). Inside the auxiliary function, the property automaton for the status variable does not transition, since the method uses its own local status variable; the rule automaton transitions to the state `st_pending` in line 10. The VSA+ESP approach merges abstract states if the property automata are in the same state, so the analysis maintains two states at the return node of the method (for the current calling context): One representing the state at the upper two exits, and one for the exit following IoMarkIrpPending. The state for the upper two exits merges the strided intervals for the return value to `1[0;0xC00002D3]`. This is a very coarse approximation of the actually possible

```
NTSTATUS FlQueueIrpToThread(PIRP Irp, ...) {
    NTSTATUS status;
    if (...) return STATUS_POWER_STATE_INVALID; // = 0xC00002D3
    if (...) {
        status = ObReferenceObjectByHandle(...);
            // SDV stub either returns STATUS_SUCCESS or STATUS_UNSUCCESSFUL
            // Joins to strided interval 0xC0000001[0;0xC0000001]
        if (!NT_SUCCESS(status)) return status;
    }
    IoMarkIrpPending(Irp);          // Property automaton transition to st_pending
    return STATUS_PENDING;          // = 0x00000103
}

NTSTATUS FloppyReadWrite(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
  NTSTATUS ntStatus;
  if (...) {
    ntStatus = STATUS_INVALID_PARAMETER;
                             //              (0xC000000D, st_not_pending,      start )
  } else {
    if (...) {
      ntStatus = FlQueueIrpToThread(Irp, ...);
                             //              (0x00000103,     st_pending,    pending)
                             //          (1[0;0xC00002D3],    st_unknown,      start ),
    } else {
      ntStatus = STATUS_SUCCESS;
                             //                     (0x0,  st_not_pending,     start )
    }
  }                          // (0xC000000D[0,0xC000000D], st_not_pending,     start ),
                             //          (1[0;0xC00002D3],    st_unknown,      start ),
                             //              (0x00000103,     st_pending,    pending)
  if ( ntStatus != STATUS_PENDING ) {
                             // (0xC000000D[0,0xC000000D], st_not_pending,     start ),
                             //          (1[0;0xC00002D3],    st_unknown,      start )
                             // Interval cannot exclude STATUS_PENDING
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
                             // (0xC000000D[0,0xC000000D], st_not_pending, completed),
                             // (1[0;0xC00002D3],             st_unknown, completed)
  }
                             // (0xC000000D[0,0xC000000D], st_not_pending, completed),
                             //              (0x00000103,     st_pending,    pending),
                             //          (1[0;0xC00002D3],    st_unknown, completed)
  return ntStatus;
}
```

Figure 6.1: Simplified code from `floppy.c` with abstract VSA/ESP states.

values `0x0` (STATUS_SUCCESS in the SDV stub for ObReferenceObjectByHandle), `0xC0000001` (STATUS_UNSUCCESSFUL in the same stub), and `0xC00002D3` (STATUS_POWER_STATE_INVALID in line 3), but it is the most precise overapproximation possible with VSA, since the greatest common divisor of `0xC0000001` and `0xC00002D3` is 1.

The abstract state holding this coarse information propagates into the dispatch function and passes the condition in line 31. The new information that the status variable is not STATUS_PENDING cannot improve the interval for ntStatus, which is another contributor to the false positive. On the call to IoCompleteRequest, the rule automaton transitions to `completed`. In the final return statement, this abstract state causes a transfer to the error state since the rule automaton is in the state `completed` and a value of STATUS_PENDING is contained in the strided interval for the return value.

Detecting Uninitialized Pointers. In `fakemodem.sys`, Jakstab encountered a potentially unsafe memory access (marked as $\times_m$). If the dispatch routine for writing (major function IRP_MJ_WRITE) is executed before the dispatch routine for opening a new connection (major function IRP_MJ_CREATE), the driver uses an uninitialized value, i.e., $(\top_R, \top_{32})$, as the index in an array write access in the driver's DeviceExtension data structure. Manual analysis of the abstract error trace confirmed the feasibility of the error trace in the execution context provided by the PnP harness. The harness only initializes devices through AddDevice and StartDevice, calls to the dispatchers for creating, reading, and writing occur nondeterministically. DDA/x86 does not check for memory safety due to the large number of false positives [8], so it did not detect this bug. As mentioned in Section 4.4, Jakstab signals an error on weak updates to all regions. This amounts to implicitly checking for write accesses to uninitialized pointers, which allowed to detect the error.

False Positives. Jakstab as well suffers from false positives from this implicit check for uninitialized pointers. This is a direct consequence of building on the

SDV harness, which is not designed for checking memory safety and often omits proper pointer allocation in the provided API stubs. False positives from write accesses to potentially uninitialized pointers are shown FP_m in Table 6.2.

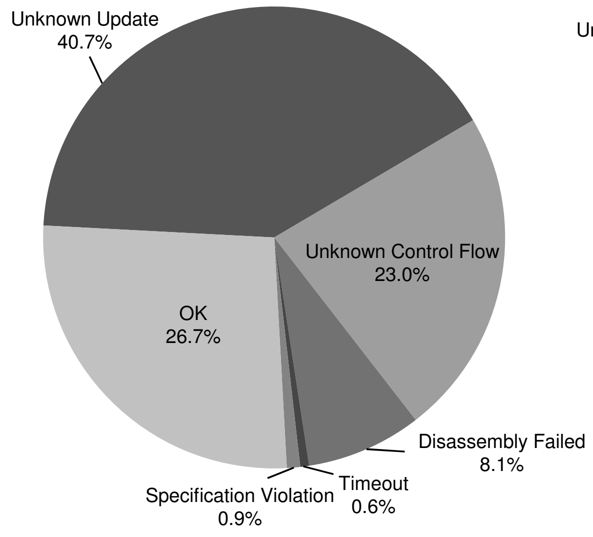
In `mouclass.sys`, a switch jump could not be resolved because the switch variable was overapproximated leading to a false positive of invalid control flow (FP_c). In the current implementation, the user has to manually investigate abstract error traces and extend the harness if necessary to eliminate false positives. A partial or full automation of this process is an area of future work.

6.1.6 Analysis of COTS Driver Binaries

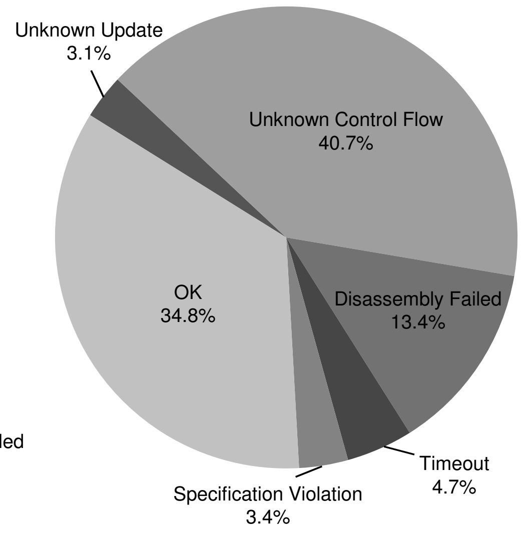
The described approach does not require to recompile and link drivers with the harness, so the experiments can be extended beyond the Windows DDK. The prototype processed all 322 drivers from the `Windows\system32\drivers` directory of a regular 32 bit Windows XP desktop system, using $k = 28$ and $k_h = 5$. Besides the *PendedCompletedRequest* rule, it also checked the *CancelSpinLock* rule, which enforces that a global lock is acquired and released in strict alternation. Note that this set of drivers also includes classes of drivers which are not even supported by the SDV harness in source code analysis, such as graphics drivers.

Nonetheless, the analysis successfully finished with verifying the specification or finding an assertion violation on 27.6% of these drivers (Figure 6.2. For 40.7%, the analysis failed because of weak global updates, mostly due to missing information about pointer allocation in the harness. In 23.0% of the cases, the analysis failed due to unknown control flow, i.e., a jump target value of $(\top_R, \top_{32})$; for 8.1%, erroneous control flow led into regions which could not be successfully disassembled. Both are caused by either side effects of API functions missing from the harness or by coarse abstraction of variables used in switch jumps. Two drivers timed out after 1 hour; in three drivers the analysis found potential assertion violations, which were not further investigated in detail.

The high number of runs failing due to weak global updates led to trying a second, unsound configuration, which still has potential to uncover bugs in



(a)



(b)

Figure 6.2: Results of analyzing 322 driver binaries from a standard Windows XP machine (a) using standard settings and (b) when ignoring weak updates.

drivers. A configuration option was added to Jakstab to simply ignore all weak updates, regardless of the region. This affects arrays, which are accessed by indexing expressions, and heap data structures, when API stubs incorrectly do not set or return pointers to them. As the properties checked are largely control oriented, though, the introduction of errors in heap data structures and arrays is less dramatic and can still find real bugs. The results for performing the experiments with this setting are shown on the right side of Figure 6.2. The outcome changed significantly with this setting; now 38.2% of the runs terminated successfully, and 4.7% timed out after an hour. Ignoring weak updates does not solve the problem of unknown or erroneous control flow, however, so now even more runs failed due to this reason (54,1%). In 3.1% of the cases, there was still a memory safety violation reported because the program appeared to free a pointer to the stack or global memory, which is most probably again due to missing information about API function behavior in the harness.

These encouraging results confirm that, using the precise approach to control flow reconstruction and static analysis described in this dissertation, it is feasible to statically analyze real world binary driver executables without access to source code. More work is needed to build better abstractions of API functions, and to streamline the process of investigating abstract error traces and eliminating false positives.

6.2 Disassembly

In manual reverse engineering or analysis scenarios where soundness is not crucial, a heuristic approach to disassembly can provide high coverage of disassembled instruction and scales even to large executables. This section presents experimental results in which Jakstab was configured to use path- and context-insensitive constant propagation (see Section 5.3.3) paired with an optimistic state transformer factory (see Section 5.2.3) and heuristics to detect additional entry points.

6.2.1 Procedure Entry Point Heuristic

The environment model, which abstracts function behavior in stubs, is only partial, due to the extremely large number of functions provided by Windows' several APIs. Therefore many callback functions that a program passes to API functions are never invoked; these functions and all functions called only by them are consequently never disassembled. If the goal of the analysis is to disassemble a maximum of instructions in large binaries, a pattern-based heuristic can help to identify additional procedure entry points in compiled code.

The following experiments employ a modified prologue in Jakstab's environment model, which was introduced in Section 5.1.4. The modified prologue not only calls the entry point of the binary, but sequentially calls *all* procedure entry points that were heuristically determined, in between resetting registers and memory values. As such, the prologue provides a simple harness that exercises the executable's code by calling each procedure from an initial state. The heuristic for finding procedures performs simple matching of byte strings: The code sequences

```
mov edi, edi
push ebp
mov ebp, esp
```

and

```
push ebp
mov ebp, esp
```

are typically used by compilers to initialize the stack frame of a new procedure. This heuristic only detects procedures using a frame pointer, but has the advantage of producing little false positives in regular compiler generated code: The shorter pattern is still three bytes long; in uniformly distributed byte strings it is expected to be encountered only after about 2^{24} bytes, or once in an executable of 16 MB.

6.2.2 Results

The set of input files to evaluate heuristic disassembly consisted of all 32 bit PE executables from the `Windows\system32` directory of a regular desktop system, 336 executables in total. The file sizes ranged from 1 KB to 2 MB, with the one outlier `MRT.exe`, which was 35 MB. Jakstab took 20 seconds on average to process the files, with a median of only 2 seconds. Due to its large size, `MRT.exe` exceeded the timeout of 20 minutes as the only file.

IDA Pro uses several heuristics to detect patterns of likely code in a binary and comes with a large collection of signatures to detect static code of common runtime libraries. This generally (for 95.5% of the executables) allows IDA Pro to disassemble more bytes than Jakstab, which was only extended with the entry point heuristic described above. On average per file, Jakstab disassembled 84%

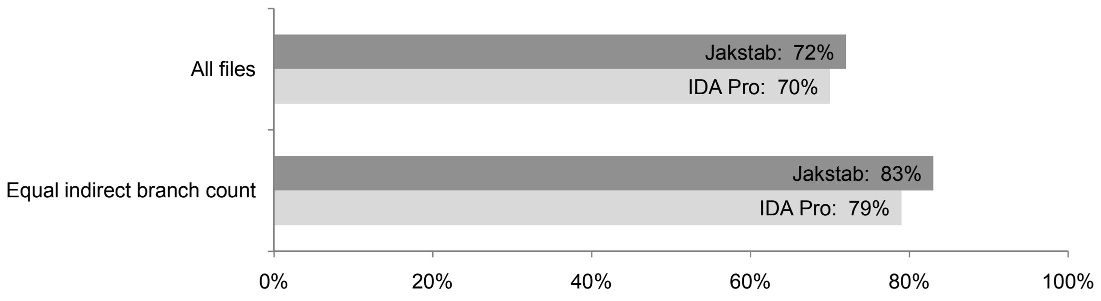


Figure 6.3: Average resolve rate of IDA Pro and Jakstab (in heuristic mode) over all executables and over only those executables where both tools reported the same total number of indirect branches.

of the instructions that IDA Pro disassembled, its set of processed instructions not necessarily being a subset of IDA Pro's, however. An absolute baseline of total instructions in an executable cannot be established without symbol information, and IDA Pro's heuristic can miss some instructions or misinterpret data as code. Adding improved heuristics for identifying function entry points (such as the machine learning based method in [120]) to Jakstab would likely increase the instruction coverage further, but possibly at the cost of false positives (data interpreted as code). The largest number of instructions processed by Jakstab was 184'819 for `MRT.exe` before timing out. Interestingly, IDA Pro disassembled only 64'347 instructions for `MRT.exe`. The largest number of instructions disassembled by IDA Pro was 521'871 instructions for `ntkrnlpa.exe`, a file on which Jakstab terminated with a runtime exception after disassembling the first 1'753 instructions.

A central point of the experiments was to evaluate Jakstab's ability to resolve indirect branch targets using solely the constant propagation domain and no detailed environment model. This inexpensive analysis mainly targets the indirect branches originating from the compiler optimization of storing the address of an imported function in a register if it is called multiple times. A similar analysis is performed by IDA Pro, which, for every such indirect branch successfully

			IDA Pro		Jakstab	
Filename	**Instructions**	**Indirect**	**Resolved**	**Time**	**Resolved**	**Time**
at.exe	4308	49	90%	1s	100%	2s
rcp.exe	2192	12	42%	1s	100%	1s
regedt32.exe	60	0	n/a	<1s	n/a	<1s
sprestrt.exe	1566	21	100%	1s	100%	1s
winver.exe	310	4	100%	<1s	100%	<1s

Table 6.4: Direct comparison of results for those executables where IDA Pro and Jakstab disassembled the same number of instructions.

resolved, adds a comment to the plain text disassembly stating the actual target. The generated assembly files were processed by a short script to count occurrences of indirect jumps and the number of resolved ones. The average over the resolve rates of all files is compared between IDA Pro and Jakstab is shown in the upper chart of Figure 6.3, and it can be seen that Jakstab slightly exceeds the capabilities of IDA Pro. This comparison is somewhat distorted, though, as the tools report different numbers of indirect jumps for many executables, due to the different number of instructions disassembled, as discussed above. For a cleaner comparison, the lower chart compares the rates only for those drivers, where the tools agreed on the same number of indirect branches (142, or 42% out of the total 336). For this set of executables, Jakstab's advantage is more pronounced.

For five executables, Jakstab and IDA Pro both disassembled exactly the same amount of instructions. Therefore, it is worth taking a closer look at these executables and to compare the results of both tools over each file individually (Table 6.4). The executables are relatively small, from less than a hundred to about 4000 instructions, and both tools disassembled each file in less than 2 seconds. One of the files did not contain any indirect branches; for the other files, Jakstab was able to successfully resolve all branches, while IDA Pro missed several on two files.

Chapter 7

Conclusions

This dissertation presents a theoretical framework for disassembly, control flow reconstruction, and static analysis on binary executables. The framework has been implemented in the novel analysis platform Jakstab, which allows the practical analysis of x86 executables using a variety of abstract domains. The usefulness of Jakstab is demonstrated through several experiments on real world binary code.

For a long time, disassembly and executable analysis were an area dominated by ad-hoc approaches, and the use of data flow analysis to augment disassembly was not cleanly formulated. Now, the knowledge that data flow guided disassembly does not suffer from a "chicken and egg" problem can help in the design of new binary analysis tools, and the provided framework can guide the development of such tools. The framework, which is defined in terms of data flow equations and constraints over an edge relation, naturally leads the way to viewing disassembly as an abstract interpretation of the binary. Control flow reconstruction then amounts to performing a reachability analysis of program counter values. The inclusion of the instruction fetch in the analysis addresses weaknesses in earlier disassembly approaches; for instance, overlapping instructions no longer cause problems since an instruction can be decoded from every memory location without alignment requirements.

A design decision that proves to be of great practical value is the use of an intermediate language. The initial cost of designing a language, implementing the translation, and populating the semantics specification is quickly amortized by the savings in maintainability, the simplicity of specifying transformers, and the portability to other architectures.

If the number of assumptions about the code is to be minimized, it is necessary to use an abstract domain for the analysis that is able to precisely represent target addresses. Overapproximating control flow targets is not a realistic option in executable code. This dissertation therefore introduces Bounded Address Tracking, which satisfies this requirement. Additionally, Bounded Address Tracking provides path sensitivity to effectively allow a context sensitive analysis without relying on the structure of procedures. Even though the analysis is expensive, a series of experiments on driver binaries shows that it is feasible on small, but real world code. In fact, it yields less false positives *and* better speed than the leading state-of-the-art approach, and it removes many of the sources of unsoundness by eliminating the separate, error-prone disassembly step.

In other analysis scenarios, where more assumptions are acceptable, the presented architecture allows to relax the control flow analysis to increase speed and to reduce the number of false positives. Using assumptions on well-behavedness of procedure calls and a simple constant propagation domain, Jakstab can perform fast disassembly in the style of regular disassemblers, and can use heuristics to increase instruction coverage even further. Experiments show that it is able to compete with the leading commercial disassembler IDA Pro using this configuration.

Future Work. Due to the high precision of Bounded Address Tracking, further improvements in scalability are a focus of ongoing and future work. An idea for an immediate next step is to investigate how the precision can be reduced where it is not required. To this end, registers and memory locations can be divided into two partitions: (i) those tracked by path sensitive Bounded Address Tracking and (ii) those overapproximated by the path insensitive strided interval analysis. For instance, high precision is required for variables that contain function pointers or return addresses. To determine the partitioning, the analysis starts by using low precision tracking for most variables. When imprecision arises in control flow, a CEGAR-like refinement loop [40] determines a new partitioning to remove spurious control flow.

Another promising direction for improving scalability is the use of summaries for sections of the control flow automaton that resemble procedures. Classical construction of function summaries exploits the structure of high level code, in which the use of procedures provides a natural partitioning. In binaries, procedures are neither explicit nor necessarily well-formed, therefore summaries have to be independent of the concept of procedures. Summaries will need to be created dynamically, whenever code is found to form a procedure-like structure. An advantage of this more general approach is that other control structures than procedures can benefit from summaries as well, e.g., switch statements that implement a parser automaton.

Self-modifying code is usually ignored in static analysis on executables, yet it is commonly encountered in practice: Every just-in-time compiler produces code at runtime, and malware uses self-modifying code to hide from virus scanners. By decoding instructions from the abstract store instead of directly from the binary, the analysis architecture can be extended to support self-modifying code. This requires a very precise analysis, since overapproximation of code bytes can produce wrong instructions and cause a cascading loss of precision. Bounded Address Tracking lends itself ideally to this purpose. Purely static analysis of self-modifying code has traditionally been deemed impossible, but by leveraging the framework devised in this dissertation, the road to achieving this goal is clearly laid out.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison Wesley, 1986.

[2] James M. Aquilina, Eoghan Casey, and Cameron H. Malin. *Malware Forensics*. Ed. by Curtis W. Rose. Syngress Publishing, 2008.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. "Dynamo: a transparent dynamic optimization system." In: *Proc. 2000 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 2000)*. 2000, pp. 1–12.

[4] Gogul Balakrishnan. Private communication. 2010.

[5] Gogul Balakrishnan. "WYSINWYX: What You See Is Not What You eXecute." PhD thesis. Madison, WI, USA: University of Wisconsin, 2007.

[6] Gogul Balakrishnan, Radu Gruian, Thomas W. Reps, and Tim Teitelbaum. "CodeSurfer/x86 – A Platform for Analyzing x86 Executables." In: *Proc. 14th Int. Conf. Compiler Construction (CC 2005)*. Ed. by Rastislav Bodík. Vol. 3443. LNCS. Springer, 2005, pp. 250–254.

[7] Gogul Balakrishnan and Thomas W. Reps. "Analyzing Memory Accesses in x86 Executables." In: *Proc. 13th Int. Conf. Compiler Construction (CC 2004)*. Ed. by Evelyn Duesterwald. Vol. 2985. LNCS. Springer, 2004, pp. 5–23.

[8] Gogul Balakrishnan and Thomas W. Reps. "Analyzing Stripped Device-Driver Executables." In: *14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. LNCS. Springer, 2008, pp. 124–140.

[9] Gogul Balakrishnan and Thomas W. Reps. "Recency-Abstraction for Heap-Allocated Storage." In: *Proc. 13th Int. Symp. Static Analysis (SAS 2006)*. Ed. by Kwangkeun Yi. Vol. 4134. LNCS. Springer, 2006, pp. 221–239.

[10] Gogul Balakrishnan, Thomas W. Reps, Nicholas Kidd, Akash Lal, Junghee Lim, David Melski, Radu Gruian, Suan Hsi Yong, Chi-Hua Chen, and Tim Teitelbaum. "Model Checking x86 Executables with CodeSurfer/x86 and WPDS++." In: *Proc. 17th Int. Conf. Computer Aided Verification (CAV 2005)*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. LNCS. Springer, 2005, pp. 158–163.

[11] Gogul Balakrishnan, Thomas W. Reps, David Melski, and Tim Teitelbaum. "WYSINWYX: What You See Is Not What You eXecute." In: *1st IFIP TC 2/WG 2.3 Conf. Verified Software: Theories, Tools, Experiments (VSTTE 2005)*. Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. LNCS. Springer, 2005, pp. 202–213.

[12] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. "Thorough static analysis of device drivers." In: *Proc. 2006 EuroSys Conf.* Ed. by Yolande Berbers and Willy Zwaenepoel. ACM, 2006, pp. 73–85.

[13] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft." In: *Proc. 4th Int. Conf. Integrated Formal Methods (IFM 2004)*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Vol. 2999. LNCS. Springer, 2004, pp. 1–20.

[14] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs." In: *Revised Lectures 4th Int. Symp. Formal Methods for Components and Objects (FMCO 2005)*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4111. LNCS. Springer, Nov. 2006, pp. 364–387.

[15] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. "Proofs from tests." In: *Proc. ACM/SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA 2008)*. Ed. by Barbara G. Ryder and Andreas Zeller. ACM, 2008, pp. 3–14.

[16] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. "A few billion lines of code later: using static analysis to find bugs in the real world." In: *Commun. ACM* 53.2 (2010), pp. 66–75.

[17] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis." In: *Proc. 19th Int. Conf. Computer Aided Verification (CAV 2007)*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. LNCS. Springer, 2007, pp. 504–518.

[18] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Program Analysis with Dynamic Precision Adjustment." In: *23rd IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2008)*. IEEE, 2008, pp. 29–38.

[19] David Binkley. "Source Code Analysis: A Road Map." In: *Workshop Future of Software Engineering (FOSE 2007)*. Ed. by Lionel C. Briand and Alexander L. Wolf. 2007, pp. 104–119.

[20] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A static analyzer for large safety-critical software." In: *Proc. ACM SIGPLAN*

2003 Conf. Programming Language Design and Implementation (POPL 2003). ACM, Jan. 2003, pp. 196–207.

[21] The Boomerang Decompiler Project. *Boomerang: A general, open source, retargetable decompiler of machine code programs*. URL: `http://boomerang.sourceforge.net/` (visited on 09/27/2010).

[22] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. "When good instructions go bad: generalizing return-oriented programming to RISC." In: *Proc. 2008 ACM Conf. Computer and Communications Security (CCS 2008)*. Ed. by Peng Ning, Paul F. Syverson, and Somesh Jha. ACM, 2008, pp. 27–38.

[23] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic." In: *Proc. 13th Int. Symp. Static Analysis (SAS 2006)*. Ed. by Kwangkeun Yi. Vol. 4134. LNCS. Springer, 2006, pp. 182–203.

[24] Sagar Chaki and James Ivers. "Software Model Checking without Source Code." In: *Proc. 1st NASA Formal Methods Symp. (NFM 2009)*. Ed. by Ewen Denney, Dimitra Giannakopoulou, and Corina S. Păsăreanu. 2009, pp. 36–45. URL: `http://ti.arc.nasa.gov/m/events/nfm09/proceedings.pdf` (visited on 09/15/2010).

[25] Sagar Chaki, Christian Schallhart, and Helmut Veith. "Verification Across Intellectual Property Boundaries." In: *19th Int. Conf. Computer Aided Verification (CAV 2007)*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. LNCS. Springer, 2007, pp. 82–94.

[26] Bor-Yuh Chang, Matthew Harren, and George Necula. "Analysis of Low-Level Code Using Cooperating Decompilers." In: *13th Int. Static Analysis Symp. (SAS 2006)*. Ed. by Kwangkeun Yi. Vol. 4134. LNCS. Springer, 2006, pp. 318–335.

[27] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. "Profile-guided Automatic Inline Expansion for C Programs." In: *Softw., Pract. Exper.* 22.5 (1992), pp. 349–369.

[28] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. "Analysis of Pointers and Structures." In: *Proc. ACM SIGPLAN'90 Conf. Programming Language Design and Implementation (PLDI 1990)*. 1990, pp. 296–310.

[29] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. "A Reachability Predicate for Analyzing Low-Level Software." In: *Proc. 13th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*. Ed. by Orna Grumberg and Michael Huth. Vol. 4424. LNCS. Springer, 2007, pp. 19–33.

[30] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. "FX!32: A Profile-Directed Binary Translator." In: *IEEE Micro* 18 (1998), pp. 56–64.

[31] M. Christodorescu and S. Jha. "Static Analysis of Executables to Detect Malicious Patterns." In: *USENIX Security Symposium*. Washington, DC, USA: USENIX Association, Aug. 2003, pp. 169–186.

[32] Mihai Christodorescu, Somesh Jha, Johannes Kinder, Stefan Katzenbeisser, and Helmut Veith. "Software transformations to improve malware detection." In: *J. Comput. Virol.* 3.4 (Nov. 2007), pp. 253–265.

[33] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Xiadong Song, and Randal E. Bryant. "Semantics-Aware Malware Detection." In: *IEEE Symp. Security and Privacy (S&P 2005)*. IEEE Computer Society, 2005, pp. 32–46.

[34] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. "String analysis for x86 binaries." In: *Proc. 2005 ACM SIGPLAN-SIGSOFT Workshop Program Analysis For Software Tools and Engineering (PASTE'05)*. Ed. by Michael D. Ernst and Thomas P. Jensen. ACM, 2005, pp. 88–95.

[35] Cristina Cifuentes and Mike van Emmerik. “Recovery of jump table case statements from binary code.” In: *Sci. Comput. Program.* 40.2-3 (2001), pp. 171–188.

[36] Cristina Cifuentes and Mike van Emmerik. “UQBT: Adaptive Binary Translation at Low Cost.” In: *IEEE Comput.* 33.3 (2000), pp. 60–66.

[37] Cristina Cifuentes and K. John Gough. “Decompilation of Binary Programs.” In: *Softw., Pract. Exper.* 25.7 (1995), pp. 811–829.

[38] Cristina Cifuentes and Shane Sendall. “Specifying the Semantics of Machine Instructions.” In: *Int. Workshop Program Comprehension (IWPC’98)*. IEEE Computer Society, 1998, pp. 126–133.

[39] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic.” In: *Workshop Logics of Programs*. Ed. by Dexter Kozen. Vol. 131. LNCS. Springer, 1981, pp. 52–71.

[40] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-Guided Abstraction Refinement.” In: *Proc. 12th Int. Conf. Computer Aided Verification (CAV 2000)*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. LNCS. Springer, 2000, pp. 154–169.

[41] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs.” In: *Proc. 10th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. LNCS. Springer, 2004, pp. 168–176.

[42] Michael Codish, Anne Mulkers, Maurice Bruynooghe, Maria J. García de la Banda, and Manuel V. Hermenegildo. “Improving Abstract Interpretations by Combining Domains.” In: *ACM Trans. Program. Lang. Syst.* 17.1 (1995), pp. 28–44.

[43] Todd A. Cook, Paul D. Franzon, Edwin A. Harcourt, and Thomas K. Miller III. "System-Level Specification of Instruction Sets." In: *Proc. 1993 Int. Conf. Computer Design: VLSI in Computers & Processors (ICCD 1993)*. IEEE Computer Society, Oct. 1993, pp. 552–557.

[44] Patrick Cousot. "Abstract Interpretation Based Formal Methods and Future Challenges." In: *Informatics – 10 Years Back. 10 Years Ahead.* Ed. by Reinhard Wilhelm. Vol. 2000. LNCS. Springer, 2001, pp. 138–156.

[45] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In: *Conf. Rec. 4th ACM Symp. Principles of Programming Languages (POPL 1977)*. Jan. 1977, pp. 238–252.

[46] Patrick Cousot and Radhia Cousot. "Abstract Interpretation Frameworks." In: *J. Log. Comput.* 2.4 (1992), pp. 511–547.

[47] Patrick Cousot and Radhia Cousot. "Systematic Design of Program Analysis Frameworks." In: *Conf. Rec. 6th Annu. ACM Symp. Principles of Programming Languages (POPL 1979)*. ACM, Jan. 1979, pp. 269–282.

[48] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya K. Debray. "A semantics-based approach to malware detection." In: *34th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2007)*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, Jan. 2007, pp. 377–388.

[49] Manuvir Das, Sorin Lerner, and Mark Seigle. "ESP: Path-Sensitive Program Verification in Polynomial Time." In: *Proc. 2002 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 2002)*. ACM, 2002, pp. 57–68.

[50] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. "Link-time binary rewriting techniques for program compaction." In: *ACM Trans. Program. Lang. Syst.* 27.5 (2005), pp. 882–945.

[51] Thomas Dullien and Sebastian Porst. "REIL: A platform-independent intermediate representation of disassembled code for static code analysis." In: *CanSecWest*. 2009. URL: `http://www.zynamics.com/downloads/csw09.pdf` (visited on 09/03/2010).

[52] Matthew B. Dwyer, John Hatcliff, Robby, Corina S. Păsăreanu, and Willem Visser. "Formal Software Analysis Emerging Trends in Software Model Checking." In: *Workshop Future of Software Engineering (FOSE 2007)*. Ed. by Lionel C. Briand and Alexander L. Wolf. 2007, pp. 120–136.

[53] Michael Eichberg, Martin Monperrus, Sven Kloppenburg, and Mira Mezini. "Model-Driven Engineering of Machine Executable Code." In: *Proc. 6th European Conf. Modelling Foundations and Applications (ECMFA 2010)*. Ed. by Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier. Vol. 6138. LNCS. Springer, 2010, pp. 104–115.

[54] Mike van Emmerik and Trent Waddington. "Using a Decompiler for Real-World Source Recovery." In: *11th Work. Conf. Reverse Engineering (WCRE 2004)*. IEEE Computer Society, 2004, pp. 27–36.

[55] Ansgar Fehnker, Ralf Huuck, Felix Rauch, and Sean Seefried. "Some Assembly Required - Program Analysis of Embedded System Code." In: *8th IEEE Int. Working Conf. Source Code Analysis and Manipulation (SCAM 2008)*. IEEE, 2008, pp. 15–24.

[56] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. "Reliable and Precise WCET Determination for a Real-Life Processor." In: *1st Int. Workshop Embedded Software (EMSOFT 2001)*. Ed. by Thomas A. Henzinger and Christoph M. Kirsch. Vol. 2211. LNCS. Springer, 2001, pp. 469–485.

[57] Halvar Flake. "Structural Comparison of Executable Objects." In: *Proc. GI SIG SIDAR Workshop Detection of Intrusions and Malware & Vulnerabil-*

ity Assessment (DIMVA 2004). Ed. by Ulrich Flegel and Michael Meier. Vol. 46. Lecture Notes in Informatics. GI, 2004, pp. 161–173.

[58] The GCC Team. *GCC, The GNU Compiler Collection*. URL: `http://gcc.gnu.org/` (visited on 08/29/2010).

[59] Patrice Godefroid and Johannes Kinder. "Proving memory safety of floating-point computations by combining static and dynamic program analysis." In: *Proc. 19th Int. Symp. Software Testing and Analysis (ISSTA 2010)*. Ed. by Paolo Tonella and Alessandro Orso. ACM, 2010, pp. 1–12.

[60] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." In: *Proc. Network and Distributed System Security Symp. (NDSS 2008)*. The Internet Society, 2008.

[61] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. "Compositional may-must program analysis: unleashing the power of alternation." In: *Proc. 37th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2010)*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, Jan. 2010, pp. 43–56.

[62] Denis Gopan and Thomas W. Reps. "Low-Level Library Analysis and Summarization." In: *19th Int. Conf. Computer Aided Verification (CAV 2007)*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. LNCS. Springer, 2007, pp. 68–81.

[63] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. "SYNERGY: a new algorithm for property checking." In: *Proc. 14th ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE 2006)*. Ed. by Michal Young and Premkumar T. Devanbu. ACM, 2006, pp. 117–127.

[64] Sumit Gulwani and Ashish Tiwari. "Combining abstract interpreters." In: *Proc. ACM SIGPLAN 2006 Conf. Programming Language Design and Implementation (PLDI 2006)*. Ed. by Thomas Ball and Michael I. Schwartzbach. 2006, pp. 376–386.

[65] Edwin A. Harcourt, Jon Mauney, and Todd A. Cook. "Functional Specification and Simulation of Instruction Set Architectures." In: *Proc. Int. Conf. Simulation and Hardware Description Languages*. SCS Press, 1994.

[66] Laune C. Harris and Barton P. Miller. "Practical analysis of stripped binary code." In: *SIGARCH Comput. Archit. News* 33.5 (2005), pp. 63–68.

[67] Klaus Havelund and Thomas Pressburger. "Model Checking JAVA Programs using JAVA PathFinder." In: *Int. J. Softw. Tools Technol. Transfer (STTT)* 2.4 (2000), pp. 366–381.

[68] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. "Lazy abstraction." In: *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation (POPL 2003)*. Jan. 2002, pp. 58–70.

[69] Hex-Rays SA. *Hex-Rays Decompiler*. URL: `http://www.hex-rays.com/decompiler.shtml` (visited on 05/12/2010).

[70] Hex-Rays SA. *IDA Pro*. URL: `http://www.hex-rays.com/idapro/` (visited on 05/12/2010).

[71] Andreas Holzer and Johannes Kinder. *Praktikum Programm- und Modellanalyse*. 2009. URL: `http://www.forsyte.de/courses/detail/index.php?id=courses.detail&arg=267` (visited on 09/25/2010).

[72] Andreas Holzer, Johannes Kinder, and Helmut Veith. "Using Verification Technology to Specify and Detect Malware." In: *Proc. 11th Int. Conf. Computer Aided Systems Theory (EUROCAST 2007)*. Ed. by Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia. Vol. 4739. LNCS. Springer, 2007, pp. 497–504.

[73] Michael Howard. *Some Bad News and Some Good News*. Oct. 2002. URL: `http://msdn.microsoft.com/en-us/library/ms972826.aspx` (visited on 08/31/2010).

[74] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. 2009.

[75] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. "F-Soft: Software Verification Platform." In: *17th Int. Conf. Computer Aided Verification (CAV 2005)*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. LNCS. Springer, 2005, pp. 301–306.

[76] Neil D. Jones. "Flow Analysis of Lambda Expressions (Preliminary Version)." In: *Proc. 8th Colloq. Automata, Languages and Programming (ICALP 1981)*. Ed. by Shimon Even and Oded Kariv. Vol. 115. LNCS. Springer, 1981, pp. 114–128.

[77] Daniel Kästner. "TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyses." In: *Proc. 2nd Int. Conf. Generative Programming and Component Engineering (GPCE 2003)*. Ed. by Frank Pfenning and Yannis Smaragdakis. Vol. 2830. LNCS. Springer, 2003, pp. 18–36.

[78] Daniel Kästner and Stephan Wilhelm. "Generic control flow reconstruction from assembly code." In: *2002 Jt. Conf. Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES'02-SCOPES'02)*. ACM, 2002, pp. 46–55.

[79] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. "Detecting Malicious Code by Model Checking." In: *Second Int. Conf. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005)*. Ed. by Klaus Julisch and Christopher Krügel. Vol. 3548. LNCS. Springer, 2005, pp. 174–187.

[80] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. "Proactive Detection of Computer Worms Using Model Checking." In: *IEEE Trans. Dependable Sec. Comput.* 7.4 (Oct. 2010), pp. 424–438.

[81] Johannes Kinder and Helmut Veith. "Jakstab: A Static Analysis Platform for Binaries." In: *Proc. 20th Int. Conf. Computer Aided Verification (CAV 2008)*. Ed. by Aarti Gupta and Sharad Malik. Vol. 5123. LNCS. Springer, 2008, pp. 423–427.

[82] Johannes Kinder and Helmut Veith. "Precise Static Analysis of Untrusted Driver Binaries." In: *Proc. 10th Int. Conf. Formal Methods in Computer-Aided Design (FMCAD 2010)*. Ed. by Roderick Bloem and Natasha Sharygina. FMCAD, Inc., Oct. 2010, pp. 43–50.

[83] Johannes Kinder, Helmut Veith, and Florian Zuleger. "An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries." In: *Proc. 10th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*. Ed. by Neil D. Jones and Markus Müller-Olm. Vol. 5403. LNCS. Springer, 2009, pp. 214–228.

[84] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. "Effective and efficient malware detection at the end host." In: *USENIX Security Symposium*. USENIX Association, 2009.

[85] Christopher Krügel, William K. Robertson, Fredrik Valeur, and Giovanni Vigna. "Static Disassembly of Obfuscated Binaries." In: *USENIX Security Symposium*. USENIX Association, 2004, pp. 255–270.

[86] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *2nd IEEE / ACM Int. Symp. Code Generation and Optimization (CGO 2004)*. IEEE Computer Society, 2004, pp. 75–88.

[87] Junghee Lim, Akash Lal, and Thomas W. Reps. "Symbolic Analysis via Semantic Reinterpretation." In: *Proc. 16th Int. Workshop Model Checking Software (SPIN 2009)*. Ed. by Corina S. Păsăreanu. Vol. 5578. LNCS. Springer, 2009, pp. 148–168.

[88] Junghee Lim and Thomas W. Reps. "A System for Generating Static Analyzers for Machine Instructions." In: *Proc. 17th Int. Conf. Compiler Construction (CC 2008)*. Ed. by Laurie J. Hendren. Vol. 4959. LNCS. Springer, 2008, pp. 36–52.

[89] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. "An Accurate Worst Case Timing Analysis for RISC Processors." In: *IEEE Trans. Software Eng.* 21.7 (1995), pp. 593–604.

[90] Cullen Linn and Saumya K. Debray. "Obfuscation of executable code to improve resistance to static disassembly." In: *Proc. 10th ACM Conf. Computer and Communications Security (CCS 2003)*. Ed. by Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger. ACM, 2003, pp. 290–299.

[91] Francesco Logozzo and Manuel Fähndrich. "On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis." In: *Proc. 17th Int. Conf. Compiler Construction (CC 2008)*. Ed. by Laurie J. Hendren. Vol. 4959. LNCS. Springer, 2008, pp. 197–212.

[92] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation." In: *Proc. ACM SIGPLAN 2005 Conf. Programming Language Design and Implementation (PLDI 2005)*. Ed. by Vivek Sarkar and Mary W. Hall. ACM, 2005, pp. 190–200.

[93] Thomas Lundqvist and Per Stenström. "Integrating Path and Timing Analysis Using Instruction-Level Simulation Techniques." In: *Proc. ACM SIGPLAN Workshop Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*. Ed. by Frank Mueller and Azer Bestavros. Vol. 1474. LNCS. Springer, 1998, pp. 1–15.

[94] Stefan Maus, Michal Moskal, and Wolfram Schulte. "Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving." In: *Proc. 12th Int. Conf. Algebraic Methodology and Software Technology (AMAST 2008)*. Ed. by José Meseguer and Grigore Rosu. Vol. 5140. LNCS. Springer, 2008, pp. 284–298.

[95] Christoph Michael. *A Feasibility Study for Static Analysis of Binary Executables*. Tech. rep. MAC-T IVV-07-203. Cigital, Inc, Dec. 2007.

[96] Microsoft Center for Software Excellence. *Binary Technologies Projects*. URL: `http://www.microsoft.com/windows/cse/bit_projects.mspx` (visited on 09/25/2010).

[97] Microsoft Corporation. *Phoenix Compiler and Shared Source Common Language Infrastructure*. URL: `http://research.microsoft.com/phoenix` (visited on 08/29/2010).

[98] Microsoft Corporation. *Windows Driver Kit documentation*. URL: `http://msdn.microsoft.com/en-us/library/ff557573(VS.85).aspx` (visited on 05/12/2010).

[99] Microsoft Corporation. *Windows Logo Program for Hardware: Overview*. URL: `http://www.microsoft.com/whdc/winlogo/default.mspx` (visited on 08/31/2010).

[100] Microsoft Corporation. *x86 Architecture*. URL: `http://msdn.microsoft.com/en-us/library/ff561502%28VS.85%29.aspx` (visited on 08/01/2010).

[101] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. "From System F to Typed Assembly Language." In: *Proc. 25th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1998)*. ACM, Jan. 1998, pp. 85–97.

[102] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Exploring Multiple Execution Paths for Malware Analysis." In: *IEEE Symp. Security and Privacy (S&P 2007)*. IEEE Computer Society, 2007, pp. 231–245.

[103] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. "Finding and Reproducing Heisenbugs in Concurrent Programs." In: *Proc. 8th USENIX Symp. Operating Systems Design and Implementation (OSDI 2008)*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 267–280.

[104] Eugene W. Myers. "Efficient Applicative Data Types." In: *Conf. Rec. 11th Annu. ACM Symp. Principles of Programming Languages (POPL 1984)*. ACM, Jan. 1984, pp. 66–75.

[105] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi cker Chiueh. "BIRD: Binary Interpretation using Runtime Disassembly." In: *4th IEEE/ACM Int. Symp. Code Generation and Optimization (CGO 2006)*. IEEE Computer Society, 2006, pp. 358–370.

[106] George C. Necula. "Proof-Carrying Code." In: *24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1997)*. ACM, Jan. 1997, pp. 106–119.

[107] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs." In: *Proc. 11th Int. Conf. Compiler Construction (CC'2002)*. Ed. by R. Nigel Horspool. Vol. 2304. LNCS. Springer, 2002, pp. 213–228.

[108] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *Proc. ACM SIGPLAN 2007 Conf. Programming Language Design and Implementation (PLDI 2007)*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, 2007, pp. 89–100.

[109] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[110] Walter Oney. *Programming the Microsoft Windows driver model*. 2nd. Redmond, WA: Microsoft Press, 2003.

[111] Oracle Corporation. *OpenJDK*. 2010.

[112] Matt Pietrek. "Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format (Part I)." In: *MSDN Magazine* 17.2 (Feb. 2002).

[113] Matt Pietrek. "Under The Hood." In: *Microsoft Systems Journal* 11.10 (Oct. 1996).

[114] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR." In: *Symp. Programming*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. LNCS. Springer, 1982, pp. 337–351.

[115] Srinivas K. Raman, Vladimir M. Pentkovski, and Jagannath Keshava. "Implementing Streaming SIMD Extensions on the Pentium III Processor." In: *IEEE Micro* 20.4 (2000), pp. 47–57.

[116] Norman Ramsey and Jack W. Davidson. "Machine Descriptions to Build Tools for Embedded Systems." In: *Proc. ACM SIGPLAN Workshop Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*. Ed. by Frank Mueller and Azer Bestavros. Vol. 1474. LNCS. Springer, 1998, pp. 176–192.

[117] Thomas W. Reps and Gogul Balakrishnan. "Improved Memory-Access Analysis for x86 Executables." In: *Proc. 17th Int. Conf. Compiler Construction (CC 2008)*. Vol. 4959. LNCS. Springer, 2008, pp. 16–35.

[118] Thomas W. Reps, Gogul Balakrishnan, Junghee Lim, and Tim Teitelbaum. "A Next-Generation Platform for Analyzing Executables." In: *3rd Asian Symp. Programming Languages and Systems (APLAS 2005)*. Ed. by Kwangkeun Yi. Vol. 3780. LNCS. Springer, 2005, pp. 212–229.

[119] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *Proc. 17th IEEE Symp. Logic in Computer Science (LICS 2002)*. IEEE Computer Society, 2002, pp. 55–74.

[120] Nathan E. Rosenblum, Xiaojin Zhu, Barton P. Miller, and Karen Hunt. "Learning to Analyze Binary Computer Code." In: *Proc. 23rd AAAI Conf. Artificial Intelligence (AAAI 2008)*. Ed. by Dieter Fox and Carla P. Gomes. AAAI Press, 2008, pp. 798–804.

[121] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Parametric Shape Analysis via 3-Valued Logic." In: *Proc. 26th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1999)*. ACM, Jan. 1999, pp. 105–118.

[122] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Solving Shape-Analysis Problems in Languages with Destructive Updating." In: *Conf. Rec. 23rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1996)*. Jan. 1996, pp. 16–31.

[123] Benjamin Schwarz, Saumya K. Debray, and Gregory R. Andrews. "Disassembly of Executable Code Revisited." In: *9th Work. Conf. Reverse Engineering (WCRE 2002)*. Ed. by Arie van Deursen and Elizabeth Burd. IEEE Computer Society, 2002, pp. 45–54.

[124] Benjamin Schwarz, Saumya K. Debray, and Gregory R. Andrews. "PLTO: A link-time optimizer for the Intel IA-32 architecture." In: *Proc. Workshop Binary Translation (WBT 2001)*. 2001.

[125] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." In: *Proc. 2007 ACM Conf. Computer and Communications Security (CCS 2007)*. Ed. by Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson. ACM, 2007, pp. 552–561.

[126] Olin Shivers. "Control-Flow Analysis in Scheme." In: *Conf. Rec. 15th Annu. ACM Symp. Principles of Programming Languages (POPL 1988)*. Jan. 1988, pp. 164–174.

[127] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. “Binary Translation.” In: *Commun. ACM* 36.2 (1993), pp. 69–81.

[128] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. “BitBlaze: A New Approach to Computer Security via Binary Analysis.” In: *Proc. 4th Int. Conf. Information Systems Security (ICISS 2008)*. Ed. by R. Sekar and Arun K. Pujari. Vol. 5352. LNCS. Springer, 2008, pp. 1–25.

[129] Amitabh Srivastava and Alan Eustace. “ATOM – A System for Building Customized Program Analysis Tools.” In: *Proc. ACM SIGPLAN'94 Conf. Programming Language Design and Implementation (PLDI 1994)*. ACM, 1994, pp. 196–205.

[130] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. “Improving the reliability of commodity operating systems.” In: *ACM Trans. Comput. Syst.* 23.1 (2005), pp. 77–110.

[131] Alfred Tarski. “A Lattice-Theoretical Fixpoint Theorem and its Applications.” In: *Pacific J. Math.* 5.2 (1955), pp. 285–309.

[132] Aditya V. Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas W. Reps. “Directed Proof Generation for Machine Code.” In: *Proc. 22nd Int. Conf. Computer Aided Verification (CAV 2010)*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Vol. 6174. LNCS. Springer, 2010, pp. 288–305.

[133] Henrik Theiling. “Extracting safe and precise control flow from binaries.” In: *7th Int. Workshop Real-Time Computing and Applications Symp. (RTCSA 2000)*. IEEE Computer Society, 2000, pp. 23–30.

[134] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Version 1.2. May 1995. URL: `http://refspecs.freestandards.org/elf/elf.pdf` (visited on 09/03/2010).

[135] Giovanni Vigna. "Static Disassembly and Code Analysis." In: *Malware Detection*. Ed. by Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Xiadong Song, and Cliff Wang. Vol. 27. Advances in Information Security. Springer, 2007. Chap. 2, pp. 19–41.

[136] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. "Static Timing Analysis for Hard Real-Time Systems." In: *Proc. 11th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2010)*. Ed. by Gilles Barthe and Manuel V. Hermenegildo. Vol. 5944. LNCS. Springer, 2010, pp. 3–22.

[137] Yichen Xie, Andy Chou, and Dawson R. Engler. "ARCHER: using symbolic, path-sensitive analysis to detect memory access errors." In: *Proc. 11th ACM SIGSOFT Symp. Foundations of Software Engineering 2003 / 9th European Software Engineering Conference (ESEC/FSE 2003)*. ACM, 2003, pp. 327–336.

[138] Greta Yorsh, Eran Yahav, and Satish Chandra. "Generating precise and concise procedure summaries." In: *Proc. 35th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2008)*. Ed. by George C. Necula and Philip Wadler. ACM, Jan. 2008, pp. 221–234.

Curriculum Vitae

Technische Universität Darmstadt, Germany 3/2008 – 11/2010

Doctoral student and research assistant in the group of Prof. Helmut Veith.

Microsoft Research, Redmond, WA, USA 6/2009 – 9/2009

Internship with Patrice Godefroid in the Research in Software Engineering (RiSE) group.

Technische Universität München, Germany 12/2005 – 2/2008

Doctoral student in the group of Prof. Helmut Veith. Scholarship by the state of Bavaria and part time research assistant.

University of Wisconsin, Madison, WI, USA 7/2005 – 9/2005

Visiting researcher in the group of Prof. Somesh Jha.

Technische Universität München, Germany 10/1999 – 5/2005

Master's degree in Computer Science (Diplom-Informatiker).

Center for Digital Technology and Management (CDTM), Munich, Germany 4/2002 – 6/2004

Certificate in Technology Management in an interdisciplinary entrepreneurship program of Ludwig-Maximilians-Universität München and Technische Universität München.