

Air Force Institute of Technology

**AFIT Scholar**

---

Theses and Dissertations

Student Graduate Works

---

3-11-2011

## Automated Analysis of ARM Binaries using the Low-Level Virtual Machine Compiler Framework

Jeffrey B. Scott

Follow this and additional works at: <https://scholar.afit.edu/etd>



Part of the [Hardware Systems Commons](#)

---

### Recommended Citation

Scott, Jeffrey B., "Automated Analysis of ARM Binaries using the Low-Level Virtual Machine Compiler Framework" (2011). *Theses and Dissertations*. 1427.

<https://scholar.afit.edu/etd/1427>

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact [richard.mansfield@afit.edu](mailto:richard.mansfield@afit.edu).



**AUTOMATED ANALYSIS OF ARM BINARIES USING THE LOW-  
LEVEL VIRTUAL MACHINE COMPILER FRAMEWORK**

THESIS

Jeffrey B. Scott, Captain, USAF  
AFIT/GCO/ENG/11-14

DEPARTMENT OF THE AIR FORCE  
AIR UNIVERSITY

***AIR FORCE INSTITUTE OF TECHNOLOGY***

---

---

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the United States Government.

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT/GCO/ENG/11-14

AUTOMATED ANALYSIS OF ARM BINARIES USING THE LOW-LEVEL  
VIRTUAL MACHINE COMPILER FRAMEWORK

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Jeffrey B. Scott, B.S.E.E.

Captain, USAF

March 2011

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

AUTOMATED ANALYSIS OF ARM BINARIES USING THE LOW-LEVEL  
VIRTUAL MACHINE COMPILER FRAMEWORK

Jeffrey B. Scott, B.S.E.E.  
Captain, USAF

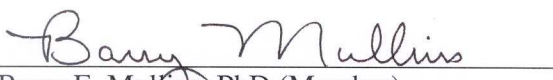
Approved:



Rusty O. Baldwin, PhD (Chairman)

3/7/11


Date



Barry E. Mullins, PhD (Member)

7 Mar 11

Date



William B. Kimball (Member)

7 Mar 11

Date

**Abstract**

Binary program analysis is a critical capability for offensive and defensive operations in Cyberspace. However, many current techniques are ineffective or time-consuming and few tools can analyze code compiled for embedded processors such as those used in network interface cards, control systems and mobile phones.

This research designs and implements a binary analysis system, called the Architecture-independent Binary Abstracting Code Analysis System (ABACAS), which reverses the normal program compilation process, lifting binary machine code to the Low-Level Virtual Machine (LLVM) compiler's intermediate representation, thereby enabling existing security-related analyses to be applied to binary programs. The prototype targets ARM binaries but can be extended to support other architectures. Several programs are translated from ARM binaries and analyzed with existing analysis tools. Programs lifted from ARM binaries are an average of 3.73 times larger than the same programs compiled from a high-level language (HLL). Analysis results are equivalent regardless of whether the HLL source or ARM binary version of the program is submitted to the system, confirming the hypothesis that LLVM is effective for binary analysis.

## **Acknowledgements**

First I would like to thank my research advisor, Dr. Rusty Baldwin, for giving me the freedom to pursue research suited to my interests and skills, for reining me in when my plans became too extravagant and for always being willing to give his time and expertise to answer questions and provide useful and detailed feedback.

I also want to thank my committee members for their help: Dr. Mullins, for teaching me how to identify weaknesses in a system by examining it from an attacker's perspective, and Mr. Kimball for his stimulating discussions, for pushing me to investigate static program analysis and for encouraging me to "learn everything there is to know about ARM."

Finally, this research would not have been possible without the support of my wife, who handled things marvelously at home and cared for me and our children while I completed this research. Her love, faith and understanding have been a constant source of motivation and strength for me.

## Table of Contents

Table of Contents .....	vi
List of Figures .....	ix
List of Tables.....	xi
I. Introduction.....	1
1.1 Problem Background.....	2
1.2 Research Goals .....	4
1.3 Document Outline .....	5
II. Literature Review .....	6
2.1 Taxonomy of Mobile Device Vulnerabilities.....	6
2.1.1 Scenario.....	7
2.1.2 Policy.....	7
2.1.3 Design.....	9
2.1.4 Implementation.....	10
2.1.5 Configuration.....	11
2.2 Security-oriented Program Analysis.....	11
2.2.1 Black Box Analysis .....	11
2.2.2 White/Grey Box Analysis.....	12
2.2.3 Dynamic Approaches .....	12
2.2.4 Static Approaches.....	15
2.3 Compiler Overview .....	18
2.3.1 The Low-Level Virtual Machine.....	20
2.3.2 Mid-end Program Analysis and Transformation .....	21
2.3.3 Back-end Code Generation.....	25
2.4 ARM Architecture Overview .....	26
2.4.1 System Programming .....	26
2.4.2 Memory Architecture .....	28
2.4.3 Instruction Set Architecture.....	29
2.5 Summary .....	32
III. System Design and Implementation.....	33
3.1 System Overview.....	33



3.2	Front-end Architecture .....	34
3.2.1	Object File Parser .....	34
3.2.2	Disassembler.....	35
3.2.3	Assembly Parser .....	35
3.2.4	LLVM IR Code Generator .....	35
3.3	Prototype Implementation .....	36
3.3.1	Object File Parser .....	36
3.3.2	Disassembler.....	37
3.3.3	Assembly Parser .....	37
3.3.4	LLVM IR Code Generator .....	41
IV.	Experimental Methodology .....	48
4.1	Approach .....	48
4.2	System Boundaries .....	50
4.3	System Services.....	51
4.3.1	Code abstraction .....	51
4.3.2	Code analysis.....	52
4.3.3	Code redocumentation.....	52
4.3.4	Code restructuring .....	53
4.3.5	Code reengineering.....	53
4.4	Workload .....	53
4.5	Performance Metrics .....	54
4.5.1	Code abstraction metrics .....	54
4.5.2	Code analysis and restructuring metrics .....	54
4.5.3	Code redocumentation metrics .....	55
4.5.4	Code reengineering metrics .....	55
4.6	System Parameters.....	55
4.7	Workload Parameters .....	56
4.8	Factors .....	57
4.8.1	“Combine instructions” transformation (-instcombine) .....	57
4.8.2	Control-flow constructs .....	58
4.8.3	Source language.....	58
4.9	Evaluation Technique.....	59

4.9.1	Phase I: Validating the model.....	59
4.9.2	Phase II: System performance analysis .....	59
4.9.3	Experimental Configuration. ....	60
4.10	Experimental Design .....	60
4.11	Methodology Summary .....	60
V.	Results .....	62
5.1	Phase I: Validating the ARM Model .....	62
5.2	Phase II: ABACAS Performance Analysis .....	64
5.2.1	Impact of source language on LLVM program analysis .....	64
5.2.2	Impact of source language on size of generated LLVM assembly code. ....	65
5.2.3	Impact of source language on LLVM restructuring transformations .....	70
5.2.4	Impact of source language on LLVM program redocumentation .....	71
5.2.5	Impact of source language on recompilation.....	74
5.3	Summary of results.....	75
VI.	Conclusions .....	77
6.1	Research Accomplishments.....	77
6.2	Contributions .....	78
6.3	Applications of this Research.....	78
6.3.1	Automatic vulnerability discovery .....	78
6.3.1	Improved malware analysis.....	79
6.3.2	Software Maintenance .....	79
6.4	Future Work .....	80
6.4.1	Expand object file support.....	80
6.4.2	Expand support for ARM .....	81
6.4.3	Add support for other architectures .....	81
6.4.4	Incorporate LLVM passes to improve system services.....	81
6.5	Conclusion.....	82
	Bibliography .....	84

## List of Figures

Figure 1. A retargetable compiler .....	20
Figure 2. C source code of a trivial program. ....	22
Figure 3. LLVM IR of the program. ....	23
Figure 4. A non-SSA program. ....	23
Figure 5. SSA version of the program. ....	24
Figure 6. Control flow a) explicitly stated in LLVM, and b) implied in ARM assembly. .....	25
Figure 7. Organization of general-purpose registers and Program Status Register [ARM08].....	27
Figure 8. Grammar for creating an AST from assembly instructions.....	36
Figure 9. System under test: ABACAS. ....	51
Figure 10. Executing a lifted LLVM program in the lli interpreter.....	62
Figure 11. Loop and dominance frontier analyses on program lifted from ARM machine code. ....	64
Figure 12. Loop and dominance frontier analyses on same program compiled from C..	64
Figure 13. Control flow graphs of main function of the Fibonacci .....	65
Figure 14. main function of Fibonacci program compiled to LLVM IR from C source code .....	66
Figure 15. C source code of Fibonacci program.....	66
Figure 16. main function of Fibonacci program lifted from ARM code. ....	67
Figure 17. ARM assembly code of main function of Fibonacci program. ....	68
Figure 18. LLVM instructions in programs generated from C source and ARM code. ..	69
Figure 19. Views generated from the Fibonacci program compiled from C source code: a) the call graph, b) CFG for main function, c) CFG for fib function, d) dominator tree for main function, and e) dominator tree for fib function. ....	72

Figure 20. Views generated from the Fibonacci program lifted from ARM machine code:  
a) the call graph, b) CFG for main function, c) CFG for fib function, d) dominator tree for  
main function, and e) dominator tree for fib function. .... 73

Figure 21. Conditional branch compiled a) from C to LLVM, and b) from LLVM to  
ARM assembly..... 74

Figure 22. Debugging the Fibonacci program a) in an x86 debugger and b) in an ARM  
debugger..... 75

## List of Tables

Table 1. ARM condition codes [ARM08]. .....	29
Table 2. Factors and factor levels .....	58
Table 3. Branch/call/return instructions validated. ....	63
Table 4. Data-processing instructions validated. ....	63
Table 5. Load/store and load/store multiple instructions validated. ....	63
Table 6. Instructions eliminated by -instcombine transformation on programs .....	70

# AUTOMATED ANALYSIS OF ARM BINARIES USING THE LOW-LEVEL VIRTUAL MACHINE COMPILER FRAMEWORK

## I. Introduction

When President Barack Obama entered office, he was the first American president to insist on keeping his smartphone [CNN09]. A self-proclaimed BlackBerry addict, President Obama fought hard to keep his mobile device after his election, viewing it as an essential communications link to the outside world. “They’re going to pry it out of my hands,” he said in describing his battle with his security advisors [Zel09]. President Obama was required to turn in his personal BlackBerry and received an NSA-approved device in its place.

The President is not the only one struggling to overcome mobile device security issues. Employees are concerned over the power their company’s IT departments have to remote-wipe their personal cell phones to protect company proprietary information [Kas10]. iPhone owners filed a lawsuit with Apple and AT&T for using a software upgrade, iPhone 1.1.1, to disable devices that had been unlocked by owners [Kei09]. In March of 2010, the US Air Force announced changes to BlackBerry services which severely restrict the capability of the devices in an attempt to bolster security [Mil10]. These are but a few examples that demonstrate commercial users, the DoD, the highest levels of government and private citizens are all struggling with mobile device security.

As more and more people use mobile technology for sensitive applications, mobile devices become more enticing targets for software attacks. Mobile devices have

several interesting characteristics that make them valuable targets. First, they are a vast storehouse of sensitive information: email, text and voice conversations, communication history and passwords are all contained therein. Second, they can act as mobile, networked sensors. Even low-end devices sold now have cameras, every cell phone has a microphone, and many devices now have GPS receivers and even accelerometers. Finally, they link to other networks. Mobile devices are incredibly well-connected and communicate over proximity connections like Bluetooth, over physical connections (like USB) to a host PC, via wireless internet connection (Wi-Fi or Wi-Max) or on the cellular network. An adversary with control of a victim's mobile device can gain a lot of information about the victim, the victim's surroundings and could use the device to gain access to other networks it communicates with.

These issues raise many questions about mobile device security. Why are mobile phones so vulnerable to attack? How do these attacks occur? How can resources be protected against these types of attacks? How can National Defense organizations use these devices and networks to fight and win in the Cyberspace warfighting domain?

### ***1.1 Problem Background***

Whether the objective is to attack or defend software on mobile devices, analysis of the software running on those devices is key to 1) discovering weaknesses to exploit, or 2) verifying that no weaknesses are present or that the software correctly conforms to some specification—that the software is secure. The first is *vulnerability discovery* and the second, *program verification*.

Many tools and techniques have been developed that analyze software for these purposes, but most of them require the software's source code—the human-readable description of the software in a high-level language. The source code for many mobile device drivers, operating system code and applications is not publicly available. Of the top five operating systems on devices sold worldwide in the first quarter of 2010 (Symbian, Research In Motion, iPhone OS, Android and Microsoft Windows Mobile), only two are open source [Gar10]. Even these have components that are provided only in binary form. Thus, analysis of binary executables is extremely desirable.

Even when source code is available, any analysis of it may be invalid, since it does not actually execute on the processor. The source code undergoes many changes during the compilation and optimization process before being loaded into memory and executed on the hardware. These changes may unintentionally invalidate properties of the program verified at the source code level, creating a mismatch between what the programmer intended and the machine code that the hardware actually executes [BR10]. It is also possible that undesirable functionality was intentionally inserted into the program during the compilation process by an untrusted compiler [Tho84].

Various approaches to binary program analysis have been studied since the early days of computing but the field remains relatively immature and most approaches are ineffective or impractical. Some key issues in the field include theoretical limitations to binary analysis, complexity of modern computing architectures and the inability to reuse analyses on multiple computing architectures.



## 1.2 Research Goals

The primary goal of this research is to develop an architecture-independent platform for automated binary program analysis based on the Low-Level Virtual Machine (LLVM) [Lat10], a modern re-targetable, open-source compiler infrastructure. At the heart of the LLVM framework is a simple, well-specified, architecture-independent intermediate representation (IR) designed to support robust and efficient program analysis, optimization and transformation [LA04]. Many analyses are already available for code compiled to LLVM IR [SH10] and several groups have developed LLVM IR analyses specifically for program verification, program safety and vulnerability discovery [BA08][DKA05][DA06][CDE08][DKA06]. These analyses are typically applied to a program after compiling the high-level source code to LLVM IR and before generating native machine code for a particular architecture.

This research develops the architecture, design and prototype implementation for an LLVM front-end which *lifts* binary machine code to the LLVM IR enabling access to the ever-growing set of analyses and transformations available for LLVM IRs. This research uses the ARM instruction set, although implementing front-ends for other architectures is straightforward. ARM is the leading producer of embedded microprocessors for mobile handheld devices [ARM10] and is therefore a logical choice for the analysis of mobile device binaries.

Additional goals include: 1) verifying the functional correctness of translated code and 2) using existing LLVM tools to analyze lifted ARM binaries.

### ***1.3 Document Outline***

Chapter 2 provides a review of the relevant literature for this research. It also provides some background information useful in understanding later chapters. Chapter 3 presents the design and implementation of the binary analysis system. Chapter 4 is the experimental methodology for verifying the system performs as desired. Chapter 5 presents the results and data analysis of the experiments performed on the system under test. Chapter 6 provides a summary of the contributions of this research, several useful applications of the developed system, and describes future work to improve the system.

## II. Literature Review

### 2.1 *Taxonomy of Mobile Device Vulnerabilities*

There are many taxonomies which categorize various aspects of software security. Some are collections of threats [Web10][FH08][Hof07], some catalog software coding errors [HLV05][TCM05], and some list and categorize vulnerabilities [Mit11][Mitr11]. The taxonomy herein is not as detailed as those listed above, but it does capture a broad range of weaknesses in mobile devices that result in successful attacks.

Bishop [Bis03] provides a number of useful definitions relating to computer security that are summarized here to eliminate any ambiguity in the following taxonomy. Computer security is often described in terms of its principle components: *confidentiality*, *integrity*, and *availability*. Confidentiality ensures only authorized subjects are able to access information. Integrity ensures no information is modified without appropriate authorization. Availability ensures the system performs its intended function whenever that functionality is required. Anything that could breach any of these areas is a security *threat*. Any weakness in a computer system that makes it possible for a threat to actually occur is a *vulnerability*. A specification that describes what should be and should not be allowed to occur with respect to confidentiality, integrity and availability of a system is a *security policy*.

Vulnerabilities result from errors in any of four areas: in the security policy, in the design intended to satisfy the policy, in the implementation of that design in hardware or software, or in the configuration of that hardware or software.

### *2.1.1 Scenario*

To facilitate discussion of mobile device security issues and provide concrete examples, it is helpful to have a fictitious scenario to refer to. Suppose Bob has a mobile device to communicate with Alice. He can call Alice, send her text messages and receive the same types of communication from her. He also uses his mobile device to make online banking transactions. Bob does not want anyone to intercept voice or data communications between him and Alice. While he is not pleased when the cell network drops his calls, he tolerates it as a minor inconvenience. He would become extremely distraught, however, if he discovered someone had tampered with his online banking account using his password, stole his money and locked him out of his own account. Bob has certain security goals for his mobile device. He wants it to be free of any defects that could enable such a breach of security. But how can Bob know if his mobile device is secure?

### *2.1.2 Policy*

Since policy defines security for a system, it should be specified at the earliest stages of product development when requirements are being defined [McG04]. Ideally, mobile device development would include the crafting of a requirements specification describing a security policy so the device will adhere to that policy. Unfortunately, security policies are often only informally defined or simply implied. If designers satisfy

an incomplete, incorrect or inadequate policy, their system may not satisfy the actual needs of their customers (i.e., they may satisfy the wrong policy). Thus, in the eyes of the customer who has his own policy in mind, the system is vulnerable to attack.

Another vulnerability arises if a policy is unenforceable. Mobile devices operate in a complex environment of interconnected systems. Bob's mobile device might be a Nokia phone with an ARM processor running Symbian OS over Verizon's cell network. Bob's phone also relies on the security of the online banking servers he connects to. Who is ultimately responsible for defining and implementing a security policy for this "system?" This issue is captured in a *Security Policy Objective* which is a statement of intent to protect a resource from unauthorized use [Ste91]. "A security policy objective is meaningful to an organization only if the organization owns or controls the resource to be protected." All of the organizations above address security in some fashion. Some of them may have defined formal policies, but they may not be able to enforce them if they do not control the resources which their piece of the overall system depends.

Finally, when dealing with interconnected systems and systems of systems, security depends on a composition of the policies, not on policies of individual systems. "A vulnerability arises when the interfaces between any two components do not match; that is, the two components do not compose according to the meaning of composition" [Win03]. This occurs when one system makes an assumption the other fails to carry out.

Assume Bob provides feedback to a new mobile device development effort. He tells the development team he wants his sensitive voice, text and internet transactions to remain confidential. The development team formulates a policy that requires all connections to entities outside the device to be encrypted. Even so, Bob's sensitive text

messages and password information may still be vulnerable to an attacker who can gain physical access to Bob's phone.

### *2.1.3 Design*

Even if a security policy perfectly captures the desired security properties of a device, a vulnerability may still persist if the system design does not completely satisfy the policy. Security policies tend to be ambiguous—they either use a non-formal language (like English) or mathematics. But in either case these policies must be implemented via a set of mechanisms. If the policy is misunderstood or inadequately addressed in the design, a vulnerability may result.

Many of the same design challenges that plague personal computers also lead to vulnerabilities in mobile devices. For instance, although software extensibility enables easy updates and extensions in functionality, it also makes it easier for malicious functionality to be added [MM00]. This is demonstrated by any number of the mobile Trojans that pose as legitimate applications. *Skulls*, for example, is a Trojan that targets Symbian-based phones. It claims to be an extended theme manager for the Nokia 7610 smartphone but renders a device non-functional on installation [FSe09][Hof07]. Complex interactions between hardware and software components both within and outside of the device are another design challenge in mobile devices. Simply addressing security in each component individually without taking into account the entire system can lead to architecture and design vulnerabilities because of the composition problem [VM04][Win03]. Other design-level vulnerabilities arise from error handling in object-

oriented code, object sharing and trust issues, unprotected communications and incorrect or missing access control mechanisms [PM04].

#### 2.1.4 *Implementation*

Assuming no security vulnerabilities existed in the policy or design of Bob's smartphone (which is a big assumption), the phone may still be vulnerable if the hardware or software does not correctly implement the design. Software coding errors (*bugs*) receive a great deal of attention in software security. Coding errors produce the types of memory corruption errors that are exploited by self-propagating worms.

Although these implementation errors certainly exist in mobile devices [MV06][MM09][HJO08][MN08], they seem to be more difficult to exploit successfully due to certain properties of the ARM processor, security features in embedded operating systems and lack of adequate tools [Mul08][San05][BFL07]. ARM processors have separate instruction and data caches. Explicit writes to memory are only reflected in the data cache. Additionally, instructions are not executed if they are written to memory that has already been cached. ARM instructions also tend to include many zeros, making it more difficult to overflow a buffer with shellcode since any zero is interpreted as a null (the end of the string). The Symbian operating system requires *capabilities* for each privileged activity an application performs. Applications must be signed with the proper capabilities or they will not be installed on the device. Additionally, the only software debuggers available for Symbian are user-mode debuggers which have a limited ability to debug privileged code.

### *2.1.5 Configuration*

Finally, even if there are no policy, design or implementation vulnerabilities in a mobile device, it may still be vulnerable to attack if misconfigured. For example, Blackberry devices come with a personal firewall which, among other things, controls activities of third-party applications. This is a useful security mechanism but its default configuration is insecure. By default, any third-party application may access sensitive data on the device, including: email, SMS messages, personal information (such as the calendar, tasks, memos and contacts), and key store (certificates, public/private keys, etc.) [Hof07]. If Bob had this kind of firewall configuration and accidentally installed a malicious application, it could exfiltrate data from his private emails, text messages and even passwords.

## **2.2 *Security-oriented Program Analysis***

This section discusses methods used for security-oriented program analysis. It focuses more on vulnerability discovery than on formal program verification. Since many of the techniques were developed for x86-based systems, this section speaks generally about vulnerability discovery and does not focus specifically on mobile devices, although all the techniques will work equally well on mobile devices.

### *2.2.1 Black Box Analysis*

Black box analysis emulates the approach a remote attacker would employ if he had no knowledge of the interior workings of a program. To him, the program is a “black box.” The only options available to him are to supply the program with input and observe the results of the operation returned to him as output. The most basic form of



black box analysis is using the program to gain as much knowledge and understanding as possible about how it works and how it might be implemented. Information gained from this step can determine what sort of input to provide and what program behavior is abnormal or faulty. Another technique, fault injection, submits spurious data in an attempt to either crash the system or to elicit a response that gives more information about how the system might be exploited. Input may be generated manually or in an automated fashion. When automated using pseudorandom data, fault injection is called *fuzz testing* or *fuzzing* [MFS90]. Fuzz testing is used extensively for finding vulnerabilities in code, including those in mobile phone protocols [MV06][MM09][HJO08][MN08]. Fuzzing provides information about the presence of errors in a program but additional analysis is typically required to determine if such errors are exploitable.

### *2.2.2 White/Grey Box Analysis*

White box analysis, targets details of the system implementation to find security weaknesses. This may include source code review or analysis of executables through disassembly. Some approaches combine white box analysis with dynamic, black box input [HM04], which is typically referred to as grey box analysis. Several grey box techniques are described in the following sections.

### *2.2.3 Dynamic Approaches*

#### *2.2.3.1 Debugging*

Debugging is a popular grey box approach. It uses special software (a debugger) which attaches to another program. The debugger monitors and controls the attached

program. There are many commercial and freely-available debuggers for various platforms. GDB is a very popular debugger for UNIX/Linux systems. SoftIce and WinDbg are two of the more powerful debuggers for Windows/x86 programs. IDAPro, a very popular disassembler, also includes a debugger for many platforms. IDAPro versions 5.3 and later include a debugger for Symbian applications [Hex08].

#### *2.2.3.2 Dynamic Information Flow Tracking*

Dynamic Information Flow Tracking (DIFT) is a technique for tracking input through a system at runtime by *tagging* or *tainting* this data and anything derived from it in the execution path. As input data flows through the system anything resulting from use of that data (e.g., a new value calculated from it, the input data copied to another location, etc.) is also *tainted* because it could potentially exploit a vulnerability in the system. Since vulnerabilities are usually exploited through some type of malicious data provided as input to a program, DIFT can identify vulnerabilities that a malicious user can reach via program input. DIFT simplifies the process of searching for vulnerabilities by reducing the search space down to a subset of possible execution paths. It has been estimated that there are five to 50 software “bugs” for every thousand lines of source code [HM04]. Not all of these are exploitable and not all that are exploitable may be exploited remotely. Nevertheless, DIFT is a powerful technique for narrowing the search space to quickly identify dangerous vulnerabilities in a system.

Several DIFT systems were developed for a particular purpose. Some automatically detect malware at runtime and prevent its execution [DKK07][KBA02][SLZ04]. Some also automatically generate signatures for detected

malware [NS05]. Other DIFT implementations identify confidential information (CI) leaks [ME07] or CI lifetime in the system [CPG04]. BitBlaze [SBY08] is a binary analysis platform that combines DIFT and static approaches to perform binary analysis on any number of applications to include vulnerability discovery and malware analysis.

There are two basic approaches used by DIFT systems. The first relies on hardware mechanisms to taint data and propagate taint information [DKK07][SLZ04]. The second approach uses binary instrumentation through software simulation [SBY08][NS05][CPG04][KBA02]. Hardware approaches usually increase the size of storage locations (memory and/or registers) by adding bits which encode taint information about the data. They also typically modify the instruction pipeline in some way to propagate taint and restrict execution based on taint information. Simulation-based approaches use dynamic binary instrumentation (DBI) platforms which add code to the binary at runtime to enable taint tracking. Two variations of DBIs exist: copy-and-annotate (C&A) and disassemble-and-resynthesize (D&R) [NS07]. The C&A approach copies each instruction as-is, annotates its effects for use by instrumentation tools and adds instrumentation code interleaved with the original instructions. The D&R approach disassembles small chunks of code at runtime, translates them to an intermediate representation, adds instrumentation IR code and generates new machine code from the IR which is executed on the processor. Binary code instrumented into an executable to perform checks and maintain a security policy is referred to as an *inline reference monitor*.

## *2.2.4 Static Approaches*

### *2.2.4.1 Static Analysis of Source Code*

Static analysis of higher-level source code can detect common programming errors or security weaknesses such as memory corruption errors [WFB00][LE01] and race conditions [Bis96]. These checks are often done at compile time to alert the programmer of errors that should be corrected as part of the software development process. In some cases, static source code analysis includes taint tracking to identify potentially unsafe uses of untrusted user data through programmer-written compiler extensions [AE02], through the use of CQual to specify “tainted” and “untainted” type qualifiers [STF01] and by Program Query Language and pointer alias analysis [LML08]. Commercial tools for static analysis, some specifically targeting embedded applications, include Coverity Static Analysis [Cov10], Grammatech CodeSonar [Gra10] and MathWorks PolySpace [Mat10].

There are several drawbacks to vulnerability discovery using static code analysis. First, source code is often simply not available. Unless the project is open source, most designers prefer to protect their intellectual property and not distribute the source code. Second, source code is not executed. It is a representation of the desired functionality of the program in a more human-readable form. For these reasons, several groups are developing methods of statically analyzing binary programs.

### *2.2.4.2 Manual Static Binary Analysis*

By manually inspecting the binary code of a program, vulnerabilities can be found in the code that actually executes on a system. Since machine code is not directly

readable by humans, a disassembler typically parses the file, converting binary op codes and operands to their assembly-language equivalents. This assembly code is reverse-engineered to understand the functionality of the program. The reverse engineer must generally identify higher-level data structures or code constructs to understand control and data flow. Finally, the reverser begins searching for weaknesses in the program design or implementation. He may begin by looking for common API functions that are used insecurely, as is the case with many string manipulation functions. He may attempt to identify locations in the code where the user supplies input to the program and manually trace this input to see if it might be used insecurely. Several good references explain this process in detail [KLA04][Kas03][Eag08]. Manual binary analysis can be very effective, but it is very time-intensive and relies on the considerable expertise of the reverser. Static information flow analysis tools help automate this process.

#### *2.2.4.3 Static Binary Information Flow Analysis*

Static binary analysis tools typically convert machine code into a more abstract intermediate representation (IR) to simplify analysis. These tools use multiple stages, similar to a compiler framework. The front-end disassembles machine code into assembly language and a separate tool parses the assembly and translates it into the IR. The mid-end or back-end contains tools that operate on this IR to perform various analyses. This is the reverse of what most compilers do, as will be described later. By reversing the translation process, converting assembly to a more abstract IR, static information flow analysis tools are more conducive to formal program analysis.

Even so, static binary analysis tools are not nearly as prevalent as static source code analysis tools, but seem to be gaining popularity. Some compiler frameworks contain plugins or utilities to “lift” binary code to an IR. ROSE [Qui11] is an open source compiler infrastructure that incorporates a tool called BinQ [QP09] which generates ROSE IR from assembly code output from a disassembler front-end. ROSE uses a high-level abstract syntax tree IR that preserves all information encoded in the input source or binary so that the source can be *unparsed* after transformations have been made. However, this IR is closely linked to the syntax of the input program so it is source or architecture-dependent. The Phoenix [Mic11] compiler framework developed by Microsoft lifts binary instructions up to register transfer language (RTL), a low-level form of IR. However, Phoenix only works on binaries compiled with a Microsoft compiler and also requires debugging information which greatly limits its usefulness. CodeSurfer/x86, a commercial tool, lifts x86 binary code that has been stripped of debug and symbol table information to an IR for analysis [BR04]. Although Valgrind is a dynamic binary analysis tool [Val10], its open-source libraries for lifting machine code to the VEX IR can be used in static binary analysis tools. Both Vine (the static analysis component to BitBlaze) and its successor, Binary Analysis Platform (BAP) use the VEX library to convert assembly instructions into VEX IR [SBY08][BJ10]. VEX, however, does not model all of the implicit operations and side effects of machine instructions so both Vine and BAP add these details to their IRs. A formal verification approach translates binary code to recursive functions expressed in Hoare logic and then verifies these using the HOL4 theorem prover [Myr09].

Static analysis is challenging primarily due to aliasing and indirect addressing. An alias occurs during program execution when two or more variables in the program refer to the same location in memory [Lan92]. Two variables *may alias* if there exists some point in a valid execution of a program that they point to the same location. Two variables *must alias* if they point to the same location for all valid executions of the program. In 1992, Landi proved the undecidability of intraprocedural *may alias* and the uncomputability of intraprocedural *must alias* for languages with dynamically allocated recursive data structures [Lan92]. Alias analysis approaches must, therefore, make simplifying assumptions to make analysis tractable [HBC99][And94]. Indirect addressing makes static disassembly even more difficult because registers used to calculate addresses can have many values.

One approach to simplify the aliasing problem for binary programs is value set analysis (VSA). VSA uses abstract data objects called *abstract locations*, or *a-locs* to represent valid locations where a variable could be stored [BR04]. These include any storage locations that are known statically—global variables defined in the data segment, local variables defined as a specific offset from the base of a stack frame and registers. VSA provides an over-approximation of the set of values an a-loc can hold at any point during execution. CodeSurfer/x86, Vine, and BAP all use VSA to calculate indirect jumps and for alias analysis [BR04][BJ10].

### **2.3 *Compiler Overview***

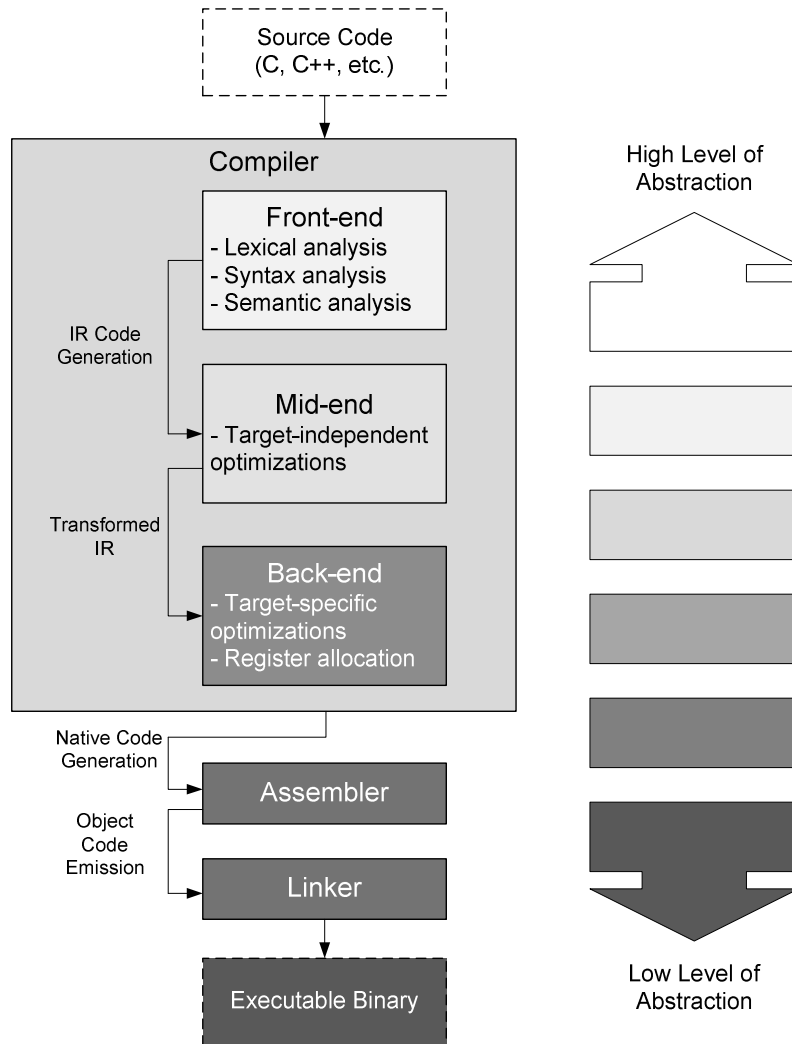
Compilers allow software to be developed at an abstract level by converting programs written in high-level languages to low-level machine code. This frees the

developer to reason and express his/her thoughts and designs in abstract terms without having to worry about details of a particular computer hardware architecture. For example, in Object-Oriented programming, programmers design their software in terms of the objects in their program, the actions those objects will perform and the relationships between objects. Compilers convert the abstract description of the program to a machine-specific implementation described in the target machine's native instruction set.

In modern compiler design, this translation process is performed modularly in successive phases [WM95][ASU88], as shown in Figure 1. The compiler front-end reads the source file as a sequence of characters, decomposes these into the various symbols or tokens used by the language (lexical analyzer/screener), parses these tokens to capture the syntactic structure of the code (syntax analysis) and converts the parsed code into an intermediate representation (intermediate code generation). Architecture-independent optimizations are performed on the IR in the mid-end. The back-end generates architecture-specific machine code from the optimized IR.

This modular design allows a high degree of programming flexibility and design re-use. Front-ends can be designed for many different source languages that all output code in the same intermediate representation. Once in this form, they can take advantage of all the transformations, analyses and optimizations written for the IR. Target-specific back-ends handle all the complexities and intricacies of modern computing architectures. Modularity eliminates the need to rewrite the entire toolchain anytime new functionality is desired whether that functionality is a new source language, a new optimization, or support for a new machine architecture.





**Figure 1. A retargetable compiler**

### 2.3.1 The Low-Level Virtual Machine

The Low-Level Virtual Machine (LLVM) is a modern retargetable compiler that focuses on mid-end transformation and back-end code generation while making it easy for front-end designers to generate LLVM intermediate code. LLVM uses a very simple instruction set of approximately 50 instructions which capture the key operations of ordinary processors. All instructions obey strict type rules and use a load/store architecture.

### 2.3.2 *Mid-end Program Analysis and Transformation*

#### 2.3.2.1 *LLVM Intermediate Representation*

LLVM intermediate representation (IR) uses an abstract RISC-type instruction set consisting of approximately 50 instructions with an infinite set of virtual typed registers in static single assignment (SSA) form. The IR may be represented in three different ways: 1) as a human-readable “assembly” language, 2) as executable bytecode on disk or 3) as an internal representation in memory, suitable for performing compiler optimizations. The internal representation is structured into `Module`, `Function`, `BasicBlock` and `Instruction` instances. A `Module` represents an entire program compilation unit. It contains a list of global variables, a list of `Functions`, a list of libraries (i.e., other `Modules` that the module depends on), a global value symbol table and a data type symbol table. A `Function` consists of `BasicBlocks`, a list of `Arguments` and a symbol table of local values. A `BasicBlock` is a list of `Instructions`.

Figure 2 shows the C source code of a trivial program which reads a value from `stdin`, assigns a 0 or a 1 to variable `A` based on the value read, adds 2 to `A` and prints the result. Figure 3 is the LLVM IR assembly module of the program. It consists of two global strings used by the `scanf` and `printf` function calls respectively (`@.str` and `@.str1`), one `main` function defined within the module, and declarations for the two externally defined functions, `scanf` and `printf`. The `main` function consists of four basic blocks: `entry`, `bb`, `bb1`, and `bb2`. Each basic block is a sequence of one or more

instructions which ends in a *terminator* instruction (i.e., a branch or a return). Each instruction which *defines* a value is in the form:

$$\langle \text{unique register name} \rangle = \langle \text{instruction mnemonic} \rangle \langle \text{operands} \rangle$$

For example, the instruction `%X = alloca i32` allocates a 32-bit integer on the local stack and assigns the address of the location to a register named `X`.

SSA form only allows a register to be assigned, or *defined*, once [CFR91] and each definition must *dominate* all *uses* of the register. In a control flow graph, one node, `X`, *dominates* another node `Y` if “`X` appears on every path from [the entry node] to `Y`” [CFR91]. `X` and `Y` may be the same node (i.e., `X` can dominate itself). “If `X` *dominates* `Y` and `X`  $\neq$  `Y`, then `X` *strictly dominates* `Y`” [CFR91]. The *dominance frontier* of a node `X` is the set of nodes `Z` in a control flow graph where `X` dominates an immediate predecessor of `Z` but `X` does not strictly dominate `Z`. Dominance frontiers identify the nodes which may require  $\phi$ -functions in SSA form. A  $\phi$ -function is placed at the beginning of a join node to select the value of an incoming variable depending on which branch control arrives from. LLVM implements SSA  $\phi$ -functions with a `phi` instruction (cf. Figure 3: the first instruction in `bb2` is a `phi` instruction). SSA form makes dataflow explicit by exposing *def-use* information through variable renaming and  $\phi$ -functions.

```
#include <stdio.h>

int main() {
    int X, A;

    scanf("%d", &X);
    if (X > 0)
        A = 0;
    else
        A = 1;
    A = A + 2;
    printf("Value of A: %d\n", A);
    return A;
}
```

**Figure 2. C source code of a trivial program.**

```

; ModuleID = 'ssa.ll'

@.str = private constant [3 x i8] c"%d\00", align 1
@.str1 = private constant [16 x i8] c"Value of A: %d\0A\00", align 1

define i32 @main() nounwind {
entry:
  %X = alloca i32
  %0 = call i32 @__isoc99_scanf(
    i8* noalias getelementptr inbounds ([3 x i8]* @.str,
    i32 0, i32 0), i32* %X) nounwind
  %1 = load i32* %X, align 4
  %2 = icmp sgt i32 %1, 0
  br i1 %2, label %bb, label %bb1

bb:
  br label %bb2

bb1:
  br label %bb2

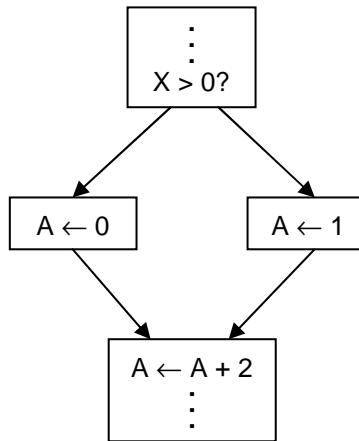
bb2:
  %A.0 = phi i32 [ 0, %bb ], [ 1, %bb1 ]
  %3 = add nsw i32 %A.0, 2
  %4 = call i32 @__printf(
    i8* noalias getelementptr inbounds ([16 x i8]* @.str1,
    i32 0, i32 0), i32 %3) nounwind
  ret i32 %3
}

declare i32 @__isoc99_scanf(i8* noalias, ...) nounwind
declare i32 @__printf(i8* noalias, ...) nounwind

```

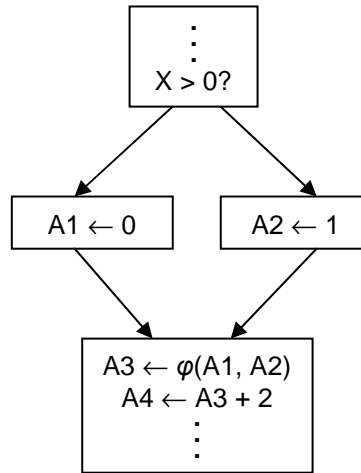
**Figure 3. LLVM IR of the program.**

The C source code in Figure 2 is not in SSA form since variable *A* is defined more than once and neither definition of *A* in the two branches dominates the use of *A* in the join node. This is easier to see in Figure 4, the control flow graph of the program.



**Figure 4. A non-SSA program.**

Figure 5 shows the SSA version of the program obtained by giving all variable definitions unique names and inserting a  $\phi$ -function to *choose* the appropriate value of  $A$  depending on which branch of the graph control is transferred from.



**Figure 5. SSA version of the program.**

LLVM enables efficient program analysis by making everything explicit in the language: dataflow is explicit (via SSA form), type information is explicit (all type conversions are explicit through several cast instructions), memory accesses are explicit (all memory accesses occur through typed pointers), even control flow is explicit since all *terminator* instructions explicitly state their *successors*. In many assembly languages, only one successor of a basic block is stated explicitly for conditional branches (cf. Figure 6b). If the condition is not satisfied, then control flow *falls through* implicitly to the next instruction. In LLVM both branch destinations are explicitly stated. In Figure 6a, control will transfer to %bb2 if the value of register %4 is true and to %bb3 if %4 is false.

<pre>%4 = icmp sgt i32 %3, 2 br i1 %4, label %bb2, label %bb3</pre>	<pre>cmp    r0, #3 blt    .LBB0_4</pre>
a)	b)

**Figure 6. Control flow a) explicitly stated in LLVM, and b) implied in ARM assembly.**

LLVM is a strongly-typed IR. The type system is low-level and language-independent and includes primitive data types of void, integer, floating-point and bool and derived types. Derived types include pointers, arrays, structures and functions. High-level or language-specific types are implemented using this low-level type system. Weakly-typed source languages must declare type information to generate valid LLVM code. This declared type information may not be reliable but it makes type information, even for weakly-typed languages, explicit. Any type conversions must be performed explicitly through LLVM cast instructions.

### 2.3.3 Back-end Code Generation

As part of native code generation, LLVM IR is lowered to an abstract architecture-specific representation of machine code consisting of `MachineFunction`, `MachineBasicBlock` and `MachineInstr` instances. Similar to their LLVM IR counterparts, `MachineFunctions` are lists of `MachineBasicBlocks`, which are lists of `MachineInstrs`. However, `MachineInstrs` contain detailed architecture-specific information about instructions including the instruction opcode, a list of operands, a list of memory operands for instructions that reference memory, and a `TargetInstrDesc` reference which encodes many of the instruction details including the instruction type (e.g., branch, call, return, etc.), instruction format, and addressing mode. This back-end

machine code representation supports valid SSA form but can also be represented in non-SSA form for manipulation after register allocation has been performed.

## 2.4 *ARM Architecture Overview*

Since ARM leads the market in processors for mobile computing [ARM10], this section briefly covers some core features of the ARM architecture. The information in this section is primarily drawn from the *ARM Architecture Reference Manual for ARMv7-A and ARMv7-R* [ARM08] unless cited otherwise.

### 2.4.1 *System Programming*

Three concepts are central to understanding the ARM architecture from a system-level perspective: privilege, mode and state. *Privilege* is the level and type of access to system resources allowed in the current state. The ARM architecture provides two privilege levels, *privileged* and *unprivileged*. *Mode* is the set of registers available combined with the privilege of the executing software. The ARM architecture supports a *user mode*, a *system mode*, and up to six exception modes as shown in Figure 7. User mode is unprivileged, all other modes are privileged. *State* is the current configuration of the system with respect to the instruction set currently being executed (Instruction Set State), how the instruction stream is being decoded (Execution State), whether or not Security Extensions are currently implemented (Security State), and whether or not the processor is being halted for debug purposes (Debug State).

An ARM processor supports up to four different instruction sets simultaneously. The processor can switch states between any one of the four as it is executing to leverage the benefits that each can provide. The four instruction set states include ARM, Thumb,

Jazelle and ThumbEE. Originally, the ARM architecture was designed as a 32-bit, word aligned RISC architecture. ARM still supports this original instruction set, with some modifications, but to increase efficiency and reduce code size, which is important in many embedded systems applications, a separate instruction set dubbed Thumb, was developed. Thumb instructions are either 16 or 32 bits aligned on a 2-byte boundary. ARM and Thumb instructions are encoded differently but implement much of the same functionality. In the Jazelle state, ARM executes Java bytecodes as part of a Java Virtual Machine (JVM). The ThumbEE state is similar to the Jazelle state, but more generic. It supports a variant of the Thumb instruction set that minimizes code size overhead by using a Just-In-Time (JIT) compiler.

System level views								
Application level view	Privileged modes							
	Exception modes							
	User mode	System mode	Supervisor mode	Monitor mode ‡	Abort mode	Undefined mode	IRQ mode	FIQ mode
R0	R0_usr							
R1	R1_usr							
R2	R2_usr							
R3	R3_usr							
R4	R4_usr							
R5	R5_usr							
R6	R6_usr							
R7	R7_usr							
R8	R8_usr							R8_fiq
R9	R9_usr							R9_fiq
R10	R10_usr							R10_fiq
R11	R11_usr							R11_fiq
R12	R12_usr							R12_fiq
SP	SP_usr		SP_svc	SP_mon ‡	SP_abt	SP_und	SP_irq	SP_fiq
LR	LR_usr		LR_svc	LR_mon ‡	LR_abt	LR_und	LR_irq	LR_fiq
PC	PC							
APSR	CPSR							
			SPSR_svc	SPSR_mon ‡	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

‡ Monitor mode, and the associated banked registers, are implemented only as part of the Security Extensions

**Figure 7. Organization of general-purpose registers and Program Status Register [ARM08].**



The ARM architecture has 16 core registers (R0-R15) that are available to an application at any given time. Registers R0-R12 are general-purpose registers. R13 is typically a *Stack Pointer* (SP), R14 is a *Link Register* (LR) and R15 is the *Program Counter* (PC). As Figure 7 shows, the current execution mode determines the set of ARM core registers currently in use. Duplicate copies (i.e., *banked registers*) of the SP and LR are provided in each of the exception modes. Registers R8-R12 are also banked in the FIQ mode to enable fast processing of interrupts. The Current Program Status Register (CPSR) contains program status and control information and is also banked across each exception mode as a Saved Program Status Register (SPSR).

Up to 16 coprocessors, CP0-CP15 extend the functionality of the ARM processor. However, CP10, CP11, CP14 and CP15 are reserved for the following purposes: CP10 is used for single-precision floating point (FP) operations and configuration/control of vector floating point (VFP) and Advanced Single-Instruction, Multiple-Data (SIMD) extensions, CP11 performs double-precision FP operations, CP14 supports debug and execution environment features, and CP15 is called the *System Control Coprocessor* since it is used for configuration/control of many features of the ARM processor system.

#### 2.4.2 Memory Architecture

From an application perspective, the ARM architecture has a flat address space of  $2^{32}$  bytes which is addressed using either a 32 bit word or 16 bit halfword alignment. The implementation details of this memory space vary depending on the ARM architecture version. ARMv7-A implements a virtual memory system architecture (VMSA) while ARMv7-P implements a simplified protected memory system architecture (PMSA). See

the *ARM Architecture Reference Manual* for more information on these implementations [ARM08].

### 2.4.3 Instruction Set Architecture

The ARM architecture supports several instruction sets but this section focuses on the core ARM instruction set. Most ARM instructions can be conditionally executed—they only perform their function if the designated condition is satisfied by one of the flags in the CPSR (cf. Table 1). The following is a basic description of the various ARM instruction categories.

**Table 1. ARM condition codes [ARM08].**

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) <sup>a</sup>	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS <sup>b</sup>	Carry set	Greater than, equal, or unordered	C == 1
0011	CC <sup>c</sup>	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) <sup>d</sup>	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.

b. HS (unsigned higher or same) is a synonym for CS.

c. LO (unsigned lower) is a synonym for CC.

d. AL is an optional mnemonic extension for always, except in IT instructions. For details see *IT* on page A8-104.

#### 2.4.3.1 Branch Instructions

Only two basic branch instructions are included in the ARM instruction set: branch to target address, *B* or a subroutine, *BL*. Either of these can also optionally change to a different instruction set (*BX* and *BLX* respectively). *B* and *BL* accept as an operand the offset of the target address from the PC value of the branch instruction. *BX* accepts the target address as a register operand. *BLX* may be executed with an immediate address or an address specified in a register.

#### 2.4.3.2 Data-Processing Instructions

Data-processing instructions include arithmetic and logical instructions, shift instructions, saturating instructions, packing and unpacking instructions, and parallel add/subtract instructions. Some of these include the option to automatically shift the second register operand by either a constant value or a register-specified value. Most can optionally update the condition flags in the CPSR register based on the result of the computation.

#### 2.4.3.3 Status Register Access Instructions

The condition flags in the CPSR are typically set during execution of data-processing instructions. However, they can be set manually with the *MSR* instruction and read manually with the *MRS* instruction.

#### 2.4.3.4 Load/Store Instructions

Load and store addresses are calculated using a base register and an offset. Three different addressing modes are possible: *offset* addressing, *pre-indexed* addressing and *post-indexed* addressing. In offset addressing, the memory address is calculated directly

by adding or subtracting the offset from the base register while the value of the base register does not change. In pre-indexed addressing, the same calculation is performed but the base register is updated with the new address to facilitate indexing through an array or memory block. Post-indexed addressing uses the base address alone as the first memory address to be accessed and updates the base address by adding or subtracting the offset. This facilitates indexing through an array or memory block.

Loads can use the PC in interesting ways. The PC can be loaded directly with the LDR instruction just as any other general-purpose register. Loads can also use the PC as the base register for an address calculation. This enables PC-relative addressing for position-independent code.

The ARM instruction set provides instructions that load or store multiple general-purpose registers at a time. Consecutive memory locations are accessed relative to a base register. Load and store multiple instructions support four different addressing modes: increment before, increment after, decrement before and decrement after. Each of these addressing modes supports updating the base register with the new value.

#### *2.4.3.5 Exception-Generating/Handling Instructions*

Three instructions provide a means for explicitly generating exceptions: the Supervisor Call (*SVC*) instruction, the Breakpoint (*BKPT*) instruction and the Secure Monitor Call (*SMC*) instruction. The main mechanism for User mode applications to call privileged code is through the *SVC* instruction.

#### *2.4.3.6 Coprocessor Instructions*

There are three types of instructions for communicating with coprocessors: coprocessor data-processing operations (CDP, CDP2), moving register values between general-purpose registers and coprocessor registers (MCR, MRC, etc.) and loading or storing coprocessor registers (LDC, STC, etc.).

### **2.5    *Summary***

This chapter provides a taxonomy of mobile device vulnerabilities, a review of the literature relevant to security-related binary program analysis and some background information on the LLVM compiler and the ARM architecture. The next chapter builds on this information and describes the binary analysis system design and implementation and the experimental methodology used to test the system.

### III. System Design and Implementation

This chapter describes the design and prototype implementation for a system which lifts binary machine code to LLVM intermediate representation. The system is called the Architecture-independent Binary Abstracting Code Analysis System, and will henceforth be referred to as ABACAS.

#### **3.1    *System Overview***

ABACAS consists of three components: a set of language-specific front-ends, an architecture-independent mid-end and a set of architecture-specific back-ends. Each front-end translates a program written in a high-level language (e.g., C++) or machine code program (e.g., ARM ELF object file) into the LLVM intermediate representation. Analyses and transformations are performed in the mid-end and the back-end transforms the IR back to a machine-code program.

This research effort develops an ARM front-end for ABACAS. The LLVM compiler framework is used without modification for the mid-end and back-end components of the system. Some of the existing mid-end analyses useful in security-related program analysis include alias analysis, pointer bounds tracking and graphical display of control flow graphs and dominator trees. LLVM back-ends currently support reliable code generation for ARM, Power PC and Intel x86 architectures [Lat10].

## 3.2 *Front-end Architecture*

### 3.2.1 *Object File Parser*

ABACAS uses the object file to determine an entry point for the instructions in the file, to retrieve all program bytes (both code and data) as a single memory buffer and to replace relocated symbolic names with the static addresses of the referenced objects in the buffer.

An object file is a static binary representation of a program. When a file is executed on a system, the operating system loader reads the object file and copies the instructions and data into memory so the processor can execute them. The object file is divided into *segments*, each containing one or more *sections* for the different types of information encoded in the file. For example, in an ELF file the text segment contains several read-only instruction and data sections including the `.text` section and the `.rodata` section. Some disassemblers use these divisions as the primary means of identifying which parts of the file to disassemble as instructions and which to treat as data. This is problematic, though, because the text segment may contain data and the data segment may contain code. An iterative approach may incorrectly disassemble data and interpret it as code and may miss sections of code which are embedded in data segments.

Object files also contain other information used during the linking process but not required by the loader to create or supplement a process image. The symbol table and string tables are examples of this. While this information is useful for analysts trying to understand binary code, *ABACAS does not depend on symbol tables, string tables, debug information or iterative disassembly of object file sections to decompile machine code to*

*LLVM IR*. This information could be incorporated to improve decompilation but the recursive-descent parsing algorithm employed by ABACAS does not require it.

### 3.2.2 *Disassembler*

Instead of using a *lexical analyzer* to return the symbols in the source document as in a typical compiler front-end, ABACAS uses a *disassembler* to scan through the raw binary input and return individual instructions, each with an assembly opcode and a set of operands. The disassembler used by ABACAS takes as input a reference to a memory object and an index into the object to begin disassembly, and returns an assembly code representation of the instruction at that location and the length of the instruction in bytes.

### 3.2.3 *Assembly Parser*

The assembly parser has a recursive-descent predictive parsing algorithm [ASU88] driven by a right-recursive context-free grammar to construct an abstract syntax tree (AST) of the program. Figure 8 shows the productions for this grammar. The parser assumes all paths through the program are executable. This may not be a correct assumption, but it is a *safe* one [Lan92].

### 3.2.4 *LLVM IR Code Generator*

Once the program has been parsed into an AST, the code generator visits each node of the AST and translates the native machine code to LLVM IR using a depth-first traversal from left to right, remembering basic blocks previously visited. Any operations of the machine code which are implicit (e.g., conditional execution, setting status flags, etc.) are made explicit in the generated LLVM code.



<i>Module</i>	<b>::=</b>	<i>Function Module</i>
	<b> </b>	$\epsilon$
<i>Function</i>	<b>::=</b>	<i>BasicBlock</i>
	<b> </b>	$\epsilon$
<i>BasicBlock</i>	<b>::=</b>	<b>returnInstr</b>
	<b> </b>	<b>indirectBrInstr</b>
	<b> </b>	<b>uncondBrInstr</b> <i>Successor</i>
	<b> </b>	<b>condBrInstr</b> <i>Successor Successor</i>
	<b> </b>	<i>InstructionSeq BasicBlock</i>
	<b> </b>	$\epsilon$
<i>Successor</i>	<b>::=</b>	<i>BasicBlock</i>
	<b> </b>	$\epsilon$
<i>InstructionSeq</i>	<b>::=</b>	<b>callInstr</b> <i>InstructionSeq</i>
	<b> </b>	<b>nonTermInstr</b> <i>InstructionSeq</i>
	<b> </b>	$\epsilon$

Figure 8. Grammar for creating an AST from assembly instructions.

### 3.3 *Prototype Implementation*

The system architecture is modular and generic enough to be used for binary programs compiled for virtually any machine architecture. However, to facilitate analysis of mobile handheld devices, the prototype system is limited to a front-end to lift binary programs compiled for the ARM architecture to LLVM.

#### 3.3.1 *Object File Parser*

Object file support in the LLVM framework is still a work in progress so IDA Pro 5.5 parses the binary file. *IDA Pro is only used as a hex editor and for fixing up relocation information. It is not used for disassembly, symbol information, or any other parsing function.* IDA Pro resolves relocation information, provides a series of program

bytes in hexadecimal as well as the index of the first byte to disassemble. There is an effort to create an LLVM API for various binary file formats including COFF, ELF, MachO [Spe10] but as yet it is not available.

### 3.3.2 *Disassembler*

The disassembler provided by the LLVM framework is used but the parsing algorithm which directs disassembly is modified. LLVM includes a command line tool, `llvm-mc` [Latt10], which is a driver for two other underlying tools—a machine code assembler and a disassembler. The machine codes are supplied as an array of hexadecimal bytes on the command line and `llvm-mc` invokes the disassembler one instruction at a time, iteratively scanning through the input. This effort adds a third tool, a *reverse* code generator, which lifts disassembled ARM code to LLVM intermediate representation. The ABACAS prototype modifies the `llvm-mc` tool so it accepts additional command line options which invoke the recursive-descent parser described in Sections 3.2.3 and 3.3.3 and to enable or disable optimization passes on the lifted LLVM IR as part of the LLVM IR code generation phase.

### 3.3.3 *Assembly Parser*

The assembly parser directs the disassembler to locations for disassembly and creates an AST of `MachineFunction`, `MachineBasicBlock` and `MachineInstr` [Lat11] objects using syntax-directed translation [ASU88]. The disassembler outputs an `MCInst` object [Latt10], which represents a machine code instruction as an *opcode* and a list of *operands* and is primarily used for textual display of assembly code. The assembly

parser converts this to the more expressive `MachineInstr` representation used in LLVM back-ends as part of the native code generation process [Lat11].

### *3.3.3.1 Parsing Basic Blocks*

One of the challenges of using syntax-directed translation of binary code is none of the syntax is explicit. A sequence of bytes must be decoded into an instruction before it can be determined how the instruction affects the structure of the program; a single instruction alone cannot provide adequate information on the structure of basic blocks. A basic block is a sequence of instructions for which the first instruction is the only entry point and the last instruction is the only exit point—all other instructions execute consecutively. The division of basic blocks in a program cannot be determined with certainty until every instruction in the program has been decoded, since any instruction could be the target of a branch, thereby marking the start of another basic block.

ABACAS uses a simplified definition of a basic block as described by the grammar in Figure 8. Initially, it considers any sequence of non-terminating instructions ending in a terminating instruction (i.e., a branch or call instruction) to be a basic block, thus ignoring the requirement that the only entry point be the first instruction. After decoding the target of a branch instruction, and before parsing the target location into a new basic block, the following procedure restructures basic blocks as necessary:

If the target location has not yet been parsed, parse the location into a new basic block.

If the target location has already been parsed and it is already the address of the first instruction in a basic block, do not parse the location again--just update the predecessors and successors lists for the parent and successor basic blocks.

Otherwise, the target address is somewhere in the middle of an existing basic block:

- Create a new basic block
- Copy all instructions from the target address to the end of the existing basic block to the new basic block
- Remove these instructions from the existing basic block
- Transfer the list of successors from the existing basic block to the new basic block
- Make the new basic block the only successor of the existing basic block
- Add the new basic block as a successor of the parent basic block (i.e., the one with the branch instruction)

### 3.3.3.2 Parsing Functions

To speak of “parsing functions” in a machine code program is somewhat of a misnomer, since functions are high-level code abstractions that do not exist in binary programs. However, Functions are a fundamental division of programs in LLVM IR. Therefore, an appropriate definition of a machine code function is created to facilitate correct translation to LLVM:

Let  $G = G(V, E)$  be a directed graph representing a machine code program where the set of vertices,  $V$ , represents all basic blocks in the program and the set of all directed edges,  $E$ , represents all control flow transfers between basic blocks.

Assume all *call* instructions (e.g., BL, BLX) return normally and are allowed in the body of a basic block (i.e., call instructions do not create edges in the graph or terminate basic blocks).

Select a vertex,  $r \in V$  such that  $r$  is the destination of a *call* instruction.

Let  $F = F(V', E')$  be a subgraph of  $G$ .

**F is a function iff  $\forall v, v \in V', v$  is *reachable* from  $r$**

### *3.3.3.3 Handling Stack References*

While not a parsing function per se, to avoid an additional AST traversal, some semantic analysis is performed by the assembly parser with respect to stack references. The `MachineFunction` class allows information about the stack frame of a machine code function to be stored via a `MachineFrameInfo` object. Every time a new instruction is parsed, it is analyzed to determine if it references the stack via the stack pointer register, SP. A virtual stack pointer is maintained throughout the parsing phase. Any time SP is incremented or decremented by an immediate value, the same operation is performed on the virtual stack pointer. This virtual stack pointer is used to calculate and record all stack references in the `MachineInstr`'s memory operand list as an offset from the value of SP on entry to the function. Each stack offset is also added to a temporary `SRefs` vector until all instructions in a function have been parsed. Before finalizing the function, the `SRefs` vector is sorted, unique and used to record all known stack information for the function in the `MachineFunction`'s `MachineFrameInfo` object.

However, this stack information is not guaranteed to be correct for two reasons: 1) if SP is modified indirectly via a register-register arithmetic operation, the value used to modify SP is unknown so it cannot be mirrored in the virtual stack pointer, and 2) if the value of SP is copied to another register, instructions could reference the stack anonymously via this proxy register. The algorithm described above does not attempt to address these issues.

### 3.3.4 *LLVM IR Code Generator*

Several goals influence the translation strategy used by ABACAS to lift ARM code to LLVM IR. LLVM is a useful representation for reverse engineering binary code in that it offers forms of its IR both in memory, for efficient analysis, and as a readable assembly language. Reverse-engineering is typically a directed endeavor where the reverser makes many decisions a computer is unable to. However, the primary goal of this research is not decompilation for human understanding, but to enable automated binary program analysis. When the goals of human readability and functional equivalence conflict, functional equivalence is chosen over readability. The guiding principle used in every aspect of translation is to model, as closely as possible, the machine instructions as they operate on data. In some cases, this results in LLVM code that seems unnecessarily verbose, but is required to accurately model the target architecture. After the initial translation is performed, simplifying transformations may be applied according to the needs of the user.

The code generator actually balances two sets of requirements: 1) those imposed by the underlying machine architecture, such as memory layout and access; and 2) those imposed by the LLVM framework, such as SSA form, functions, function arguments and strict data types. LLVM imposes high-level abstractions which are not present in the machine code, while the machine code requires accurate modeling of concrete architecture features which LLVM IR was not designed to model. While much of the translation process is straight-forward and can be performed by simply re-implementing the semantics of each ARM instruction explicitly using the LLVM instruction set, some

translation challenges arise due to conflicts between these requirements. The following sections describe how the current implementation overcomes these challenges.

#### *3.3.4.1 Generating Functions*

Before a function is generated, the code generator performs semantic analysis of the `MachineFunction` in the AST to identify arguments to the function. No attempt is made to recreate the original arguments of the function as they existed in the high-level source code. Rather, the translated LLVM function arguments are determined by how the variables are used by instructions in the function: Any register which is used in the function before being defined in the function is an argument and any reference to stack memory at an address equal to or higher than the value of SP on entry to the function is also an argument to the function, passed via memory.

Once arguments have been discovered, a `Function` object is created with an entry `BasicBlock` corresponding to the first `MachineBasicBlock` in the `MachineFunction`. Each `MachineInstr` in the basic block is translated into one or more LLVM Instructions until either a terminator instruction is reached (e.g., a branch or return) or until all instructions in the `MachineBasicBlock` are generated (e.g., the basic block falls through implicitly to its successor block). If the last instruction is a branch, code for each of the `MachineBasicBlock`'s successors is generated before completing code generation of the first `MachineBasicBlock`. Thus, the `BasicBlock` with the longest path from the entry `BasicBlock` will complete code generation first.

#### 3.3.4.2 Generating Stack References

Since many software vulnerabilities are a result of memory corruption errors, ABACAS code generation models ARM memory accesses as accurately as possible. However, LLVM uses a very abstract representation of a stack frame not well suited for this purpose. In LLVM, local variables are allocated memory via an `alloca` instruction and are freed automatically on return from the function. The allocated variables may not be contiguous or arranged in any particular order in physical memory. Furthermore, the process of lifting machine code to LLVM can complicate memory analysis since some native machine code instructions must be translated using additional `alloca` instructions. For example, LLVM does not have a `mov` instruction to transfer a value from one register to another. Instead, this is modeled by creating a stack object, via `alloca`, to hold the value, storing the value to this object, then loading the value into the new register. ABACAS translates the ARM instruction

```
mov    r0, #13
```

into the following LLVM assembly code when no optimizations are used:

```
%tmp = alloca i32
store i32 13, i32* %tmp
%R0_ = load i32* %tmp
```

To overcome this, ABACAS models the ARM stack by allocating an array of bytes, which it names `%stack_vars`, to the LLVM abstract stack using the `alloca` instruction. All memory accesses present in the native code operate on this array when translated to LLVM IR and any additional memory references are allocated with separate `alloca` instructions. The array is just large enough to hold all stack objects referenced in the machine code. These stack objects are identified during the assembly parsing stage



as described in Section 3.3.3.3. The ARM stack grows downward in memory, so any instructions which reference memory within the stack frame will first decrement SP. To model this behavior, ABACAS uses the LLVM `getelementptr` instruction to define SP with the address of the byte just past the end of the array before any other instructions in the function are translated. This allows stack references in the native ARM code to be translated directly to similar LLVM instructions operating on the array, ensuring the generated instructions have a chunk of contiguous memory that they can store to and load from.

#### 3.3.4.3 Handling arguments passed on the stack

Machine code programs use registers and memory to implement passing arguments to functions described in high-level source code. In the forward compilation process, compilers follow *procedure call standards* or *calling conventions* to implement the passing of arguments to functions but hand-coded or obfuscated assembly might not follow these conventions. The ARM application binary interface (ABI) specification [ARM09] allows arguments that are 32 bits in size or smaller to be passed via registers R0-R3. If an argument does not fit in these registers, or if there are more than 4 arguments, the value is spilled to the stack by the caller function and the callee loads the value directly from the stack location, which is at a positive (or zero) offset from the value of SP on entry to the callee function. As described in Section 3.3.4.1, *ABACAS does not depend on calling conventions to identify arguments, but relies on register def/use analysis and stack offsets instead.*

Generating LLVM code for arguments passed via registers is straight-forward, but doing this for arguments passed via memory is more challenging. After performing the semantic analysis described in Section 3.3.3.3, ABACAS knows the sizes of the local variables used in the function and knows the addresses of the arguments passed on the stack, but has no knowledge of the sizes of variables in the caller's stack passed as arguments. Although the goal is to model the native code and architecture as accurately as possible, this is difficult since stack memory is segregated between LLVM functions and not contiguous as in an ARM-based device.

ABACAS generates stack arguments by passing the address of the variable in the caller's stack as an argument to the callee function. When `stack_vars` is generated (cf. Section 3.3.4.2), four bytes are added to hold the four byte address of each argument passed on the stack and SP is set to the array index equal to the size, in bytes, of all local stack variables in the callee function. For example, if the callee function references four local variables, each four bytes in length, and two stack arguments, `stack_vars` will be 24 bytes in size and SP will initially point to the byte at index 16 (the first byte of the first argument). Any time a reference is made to a variable at a stack offset (from the value of SP on entry to the function) greater than or equal to zero, it is *dereferenced* first using an additional load instead of being used directly since the callee's stack only holds the address of the argument, not the value as it would be on a real ARM device.

#### 3.3.4.4 Type inference

Another challenge of generating LLVM IR from ARM machine code arises from the fact that LLVM IR is a strongly-typed language but machine code does not include

any explicit type information. LLVM was designed to support strongly-typed and weakly-typed languages [LA04] and includes several type conversion instructions to aid in code generation. LLVM also uses a very low-level type system designed so that a wide variety of data types can be implemented in LLVM.

Here, too, ABACAS uses a translation philosophy which makes as few assumptions about the nature of the code as possible. Data types are selected and refined according to how the machine instructions use the data. When a new type must be specified and ABACAS already knows what instructions operate on the data, it chooses the most concrete, valid data type for that instruction. In many cases, this is a 32 bit integer (LLVM type `i32`) since the ARM general-purpose registers are 32 bits in size and most data processing instructions operate on the entire value in the register. Some ARM instructions operate on other sizes, including 8, 16 or 64 bit data values (e.g., `LDRB`, `LDRH`, `LDRD` respectively), but these instructions have not been implemented in ABACAS. If the data value is known to be a memory location, then a pointer type is selected.

Once a type is selected, cast instructions are generated as necessary to meet the requirements of other instructions operating on the data. This is similar to how the machine code operates on data—an ARM instruction does not care if the data it operates on is a 32-bit integer, a 32-bit pointer to integer, or a 32-bit pointer to a character, but it does not explicitly cast the data from one type to another. In LLVM, everything must be explicit. For example, if a value is used as the destination operand of a store instruction it must be cast to a pointer if it is not a pointer already since the destination of a store instruction is a memory location.

Sometimes a data type must be generated before any information is available to aid in selecting a valid data type. Fortunately, LLVM includes an `Opaque` abstract type which allows the code generator to postpone selecting a more concrete type. When more information is known, the `Opaque` type is refined to a concrete type and every value which uses the type is updated automatically. ABACAS uses `Opaque` types for function declarations since the return type is not known until the entire function has been generated. Whenever ABACAS employs `Opaque` types, it refines them at the earliest opportunity.

## IV. Experimental Methodology

The overarching goal of this effort is to develop an architecture-independent platform for automated analysis of binary programs. The experimental goals are twofold: 1) to create a front-end for the Low-Level Virtual Machine (LLVM) compiler framework [Lat10] that correctly translates ARM machine code into the LLVM intermediate representation, and 2) to determine the effectiveness of the system for performing automatic program analyses on binary programs compiled for the ARM architecture. Existing LLVM analyses and transformations performed on LLVM IR generated from the prototype ARM front-end are expected to produce comparable results to the same analyses and transformations run on LLVM IR compiled from the high-level source code of the same program. This chapter describes the methodology for evaluating the system and thus verifying the research goals have been met.

### **4.1    *Approach***

In modern compiler design, multiple stages convert source code to architecture-specific machine code [WM95]. The front-end reads the source file as a sequence of characters, decomposes these into the various symbols or tokens used by the language, parses these tokens to capture the syntactic structure of the code and converts the parsed code into an IR. Optimizations are performed on the IR in the mid-end and the back-end generates architecture-specific machine code from the optimized IR.

This research leverages the modular design, flexible, architecture-independent IR and efficient program analysis capabilities of the LLVM compiler infrastructure to create ABACAS, an architecture-independent binary program analysis system. Specifically, the typical high-level source code front-end (e.g., `llvm-gcc`, a C-to-LLVM front end) is replaced with an ARM-to-LLVM front end and the mid-end and back-ends are used to analyze and transform the ARM binaries.

The ARM front end uses three phases to lift binary programs to LLVM IR: an object file parsing phase, an abstract syntax tree (AST) creation phase and a code generation phase. In the first phase, the object file is parsed to retrieve the necessary information for the AST creation phase. At a minimum, this includes providing a single buffer of all program bytes (both instruction and data) and the index within the buffer of the first instruction to disassemble. Relocation information must be resolved with appropriate offsets within this buffer. The AST creation phase employs a recursive-descent parsing algorithm to disassemble and parse the input buffer into an AST, beginning at the entry point supplied by the object file parsing phase. The final code generation phase performs a depth-first traversal of the AST and generates LLVM IR for each node.

After validating ABACAS translates individual ARM instructions correctly, the performance of the system is tested by measuring its response to several ARM binary programs submitted as a workload to the system. Finally the system's response to ARM binary programs is compared with response to the same programs submitted in their high-level-language formats.

## 4.2 *System Boundaries*

The system under test (SUT), dubbed the Architecture-independent Binary Abstracting Code Analysis System (ABACAS), is composed of a set of language-specific front-ends, a mid-end and a back-end as shown in Figure 9. Although part of the ABACAS back-end, the assembler and linker are shown as separate components because they are external to the LLVM back-end. The workload is shown on the left in dashed boxes as a program submitted in three different formats corresponding to three different levels of abstraction: binary object code, LLVM assembly code, and high-level source code. Lighter boxes indicate a more abstract format and darker boxes indicate a more concrete format. The dashed boxes on the right side of the diagram represent the system responses, also corresponding to different abstraction levels. The system service responses, workload and system parameters are described in Sections 4.3, 4.4, and 4.6 respectively.

The prototype currently only recognizes a subset of ARM instructions to demonstrate the viability of this approach and demonstrate its ability to perform static program analysis on binary programs lifted to the LLVM intermediate representation. This ARM-to-LLVM front-end is the component under test (CUT). The prototype lifts a subset of branch instructions, data-processing instructions and load/store instructions. Support for status register access instructions, exception-generating/handling instructions and coprocessor instructions has not been implemented. ABACAS currently supports loads and stores for stack memory; global and heap memory accesses are not implemented. Furthermore, only ARM-specific encodings of the instructions are handled. No Thumb, ThumbEE or Jazelle instructions are supported.

The SUT does not include the user of the system, testing software or target hardware (such as a smartphone or PC). Although these hardware and software components are used to verify the correct operation of the system during development, they do not provide any of the system services and are therefore excluded from the SUT.

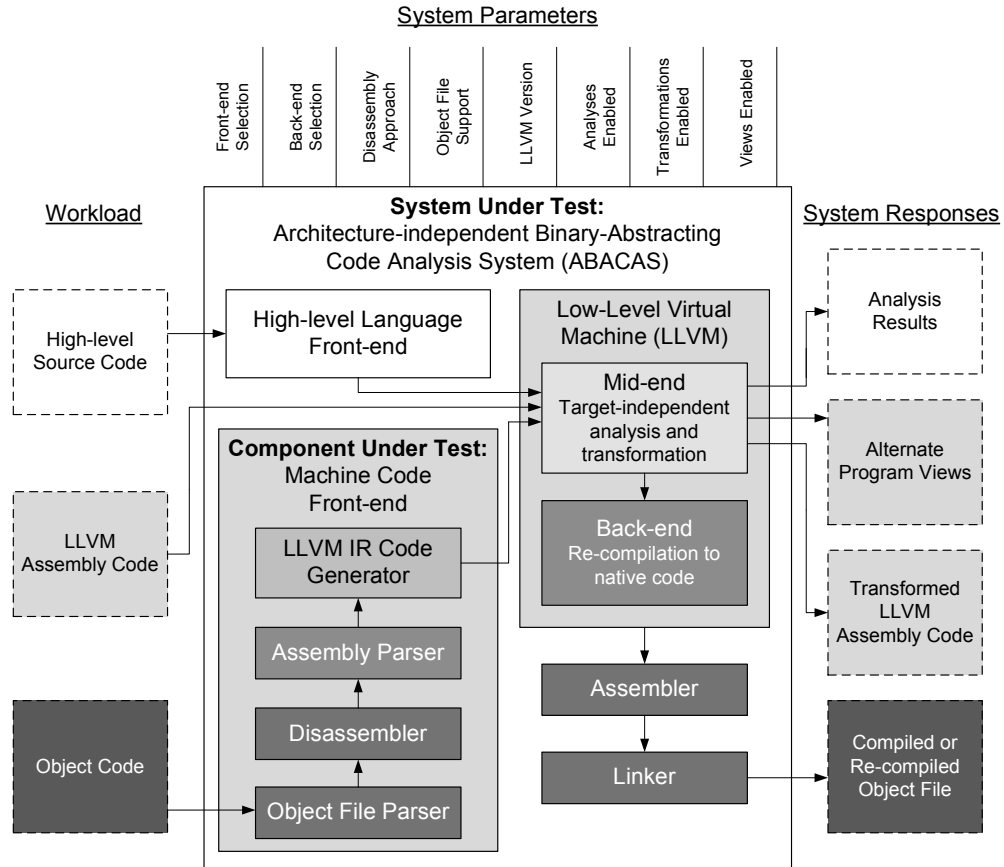


Figure 9. System under test: ABACAS.

### 4.3 System Services

ABACAS provides the following services.

#### 4.3.1 Code abstraction

This service converts code from one representation to a semantically equivalent representation at a higher level of abstraction. Two examples of this include *disassembly*



from binary machine code to ARM assembly and *reverse code generation* from ARM assembly to LLVM IR. Only reverse code generation is considered in this thesis and assumes the disassembly phase has no errors. Possible outcomes of this service are: 1) transformation successful, 2) successful with errors (representations are not semantically equivalent) and 3) unsuccessful.

#### 4.3.2 *Code analysis*

Code analysis traverses some portion of a program and computes information for use by other analysis or transformation passes, for debugging purposes, or for program visualization [Lat10]. LLVM provides many different analysis passes. Only two are tested: a loop detection pass and a pass which calculates dominance frontiers. Possible outcomes of this service include: 1) analysis completed successfully, 2) analysis completed with errors, and 3) unsuccessful. Only outcomes 1) and 3) are considered since these analysis passes are already provided by the LLVM compiler. It is beyond the scope of this effort to determine the correctness of core LLVM functionality.

#### 4.3.3 *Code redocumentation*

*Redocumentation* is “the creation or revision of a semantically equivalent representation within the same relative abstraction level” [CC90]. By lifting machine code to LLVM IR, ABACAS enables viewing the program in many different forms: as LLVM assembly language, as a control-flow graph, as a call graph and as a dominator tree. These representations improve program understanding in one way or another. Possible outcomes of this service include: 1) view created successfully, 2) view created

with errors, and 3) unsuccessful. Again, only outcomes 1) and 3) are considered since these views are already provided by the LLVM compiler.

#### *4.3.4 Code restructuring*

Code restructuring transforms code from one form to another semantically equivalent form at the same level of abstraction [CC90]. Most LLVM IR transformations fall into this category. Possible outcomes of this service include: 1) transformation completed successfully, 2) transformation completed with errors and 3) unsuccessful. Again, only outcomes 1) and 3) are considered since these are already provided by the LLVM compiler.

#### *4.3.5 Code reengineering*

Code reengineering combines reverse engineering and forward software engineering to either restructure or otherwise transform and *re-implement* code to meet new requirements [CC90]. Possible outcomes of this service include: 1) re-implementation completed successfully, 2) re-implementation completed with errors and 3) unsuccessful.

### **4.4 Workload**

ABACAS supports three different types of users at three different levels of abstraction: users submitting high-level source code, those submitting code already in LLVM IR form and those submitting ARM object code. These users may be interested in analyzing their code to gain information, in transforming their code to a different form at a higher, lower or equivalent level of abstraction or in combining code from multiple

levels of abstraction and re-compiling the new program into a native machine-code format.

## 4.5 *Performance Metrics*

Seven metrics compare performance of the system in terms of the five primary services provided by the system.

### 4.5.1 *Code abstraction metrics*

Instruction set coverage ( $\%Cov$ ): A measure of how completely the machine-to-IR front-end translation is performed. It is measured as a percentage of instructions translated out of the total number of instructions in an instruction category. Instruction categories include branch instructions, data processing instructions, status register access instructions, load/store instructions, exception generating/handling instructions and coprocessor instructions.

### 4.5.2 *Code analysis and restructuring metrics*

Loop detection rate ( $LDR$ ): A measure of the system's accuracy in detecting loops in a program, measured as a ratio of loops detected out of total number of loops in a program.

Dominance frontier detection rate ( $DDR$ ): A measure of the system's accuracy in calculating dominance frontiers in lifted and/or transformed IR, measured as ratio of dominance frontiers detected out of total number of dominance frontiers in a program.

Instructions ( $I_{Total}$ ): A count of the number of LLVM instructions in a program.

Instruction reduction rate ( $IRR$ ): A measure of the effectiveness of a program transformation at reducing code size, measured as the following ratio:

$$IRR = \frac{I_{Elim}}{I_{Total}} \quad (1)$$

where  $I_{Elim}$  is the number of instructions eliminated by the transformation and  $I_{Total}$  is the total number of instructions in the program before transformation.

#### 4.5.3 *Code redocumentation metrics*

View generation rate (VGR): A measure of the system's ability to present the code in different ways, measured as a ratio of views displayed out of views attempted.

#### 4.5.4 *Code reengineering metrics*

Recompilation success rate (RSR): A measure of the system's ability to recompile modified programs lifted from ARM binaries back to executable programs. *RSR* is measured as a ratio of the number of recompiled programs which execute correctly out of the number of programs recompiled.

### 4.6 *System Parameters*

The following parameters affect system performance:

Analysis passes enabled: Some analyses are expected to produce significantly different results when performed on the lifted IR compared to the IR generated from the original source code, while some analyses are expected to produce very similar results. For example, the lifted code is expected to have significantly higher instruction count than the original IR but to have very similar loop construction results.

Transformation passes enabled: Transformations impact the size and readability of lifted code. Restructuring transformations should not impact the semantics of the code.

Views enabled: Different views help the user understand the code.

Back-end code generators used: Backend code generators affect performance of the reengineering service by enabling recompilation to different machine architectures.

Front-end languages supported: Front-ends determine what programs can be submitted as input to the system.

Disassembly approach: Disassembly in a front-end could be performed recursively or iteratively. Both techniques have performance implications, especially in the presence of anti-disassembly features.

Object file support: Object file support determines what binary file formats the system accepts.

LLVM version: Different versions affect what analysis and optimization passes are available. Older versions also contain bugs that are fixed in newer versions.

#### **4.7 Workload Parameters**

Service requests to the SUT come in the form of programs to be analyzed. Each program has several different parameters which are described below:

Control flow constructs: Programming languages use different high-level constructs to control execution of a program. More complex control flow constructs in the source program results in more complex control flow in the machine code that is more difficult to lift to IR.

Source language: The real value of ABACAS is its ability to analyze code written in both machine languages and high-level languages. The more languages supported in the front-end, the more programs can be analyzed.

Program size: Program size is an indication of program complexity. More complex code is more difficult to analyze.

Anti-reverse engineering: Some software designers and malware writers employ an array of techniques to prevent reverse-engineering of their code. Some of these techniques include: obfuscation, self-modifying code, anti-disassembling and anti-debugging. Any of these may affect the performance of an automated reverse engineering system.

#### **4.8 Factors**

Three parameters are selected as factors to test the performance of the system. These are summarized in Table 2 and described below:

##### *4.8.1 “Combine instructions” transformation (-instcombine)*

This transformation reduces program size by combining instructions into fewer, simple instructions. This should make it easy to quantify changes in a program and compare transformations in IR generated from source code and IR lifted from ARM machine code. Although the transformations should result in semantically-equivalent programs, it is possible that they will affect the outcomes of analyses, views or the ability to recompile to native code.

This pass is expected to significantly reduce the code size of IR lifted from machine code since the ARM-to-LLVM translation strategy makes no attempt at generating code efficiently. This factor has two levels: enabled and disabled.

#### 4.8.2 Control-flow constructs

Each of the following control-flow constructs add complexity to a program. A program with all four constructs is expected to be the most difficult to translate correctly.

Conditionals: These include if-then statements in high-level languages and introduce conditional branches in machine code.

Loops: These add more complications to SSA construction.

Recursion: Recursive function calls could cause depth-first algorithms, like the parser and code generator employed by ABACAS, not to terminate if they are not implemented correctly.

#### 4.8.3 Source language

Two source languages are demonstrated, one high-level language and one machine language:

C code: Generated with the llvm-gcc front-end

ARM machine code: Generated with the ABACAS ARM-to-llvm front-end

**Table 2. Factors and factor levels**

Levels	Transformation (-instcombine)	Source Language	Control-flow Constructs
1	Disabled	C	Conditionals
2	Enabled	ARM	Loops
3			Recursion

## 4.9 *Evaluation Technique*

Two separate phases are used to evaluate performance of the system. The ARM front-end defines a model of the ARM architecture in LLVM which captures the semantics of each instruction modeled. The first phase validates this model against both a simulation and measurement of an emulated system in operation. The second phase compares performance of the system using programs translated with this model and programs translated from C source code.

### 4.9.1 *Phase I: Validating the model*

A modified version of the LLVM interpreter, lli, simulates each modeled instruction and prints the value of the instruction in a trace capture file. The results of this simulation are compared to execution of the native ARM code in a debugger on an emulated ARM processor. Corrections to the model are made as necessary.

### 4.9.2 *Phase II: System performance analysis*

Once the model has been validated, programs are submitted as a workload to the system. Workload programs written in two different source languages are submitted: programs written in C and the same programs compiled to ARM machine code. Various configurations of the system and workload parameters are tested and the results analyzed to determine if there is a significant difference between analyses performed on IR from C source code and IR from ARM machine code. Programs recompiled to a machine architecture are executed in a debugger and compared with the baseline ARM code, also executed in a debugger.



#### *4.9.3 Experimental Configuration.*

ABACAS is based on the LLVM framework and is hosted on a 2.00GHz Intel Core2 Duo CPU with 2GB of RAM running the Ubuntu 9.10 operating system. A pre-LLVM 2.8 release version of LLVM is used from the main development trunk, revision 116203. To validate the modeled ARM instructions, Qemu Manager 7.0 is used to run an emulated ARM Versatile/PB (ARM926EJ-S) processor with Ubuntu 5.0, kernel 2.6.26-2. The ARM assembly files output from LLVM's llc tool are compiled to native ARM code on the emulated ARM processor using gcc 4.3.2 and debugged using gdb 6.8. Intel x86 assembly files output from llc are compiled to native code with gcc 4.4.1 and debugged with gdb 7.0.

#### *4.10 Experimental Design*

A full factorial design is used for Phase II of this experiment. There are three factors with 2, 2 and 3 levels respectively. This results in  $2 \times 2 \times 3 = 12$  required experiments. The variance in system response is anticipated to be very low. A sufficiently narrow confidence interval at the 95% confidence level should be achieved with approximately 3 replications. Therefore, a total of  $12 \times 3 = 36$  experiments are required.

#### *4.11 Methodology Summary*

A modern optimizing compiler framework like LLVM which provides a platform-independent IR is a promising framework for platform-independent program analysis and reverse engineering. This chapter presents a methodology for determining the

effectiveness of such a framework by creating an ARM-to-LLVM front-end translator. This allows the same analyses to be performed on the IR independent of program format.

## V. Results

The performance metrics described in Section 4.5 characterize the performance of ABACAS in terms of the five system services, *code abstraction*, *code analysis*, *code redocumentation*, *code restructuring* and *code reengineering*. Code abstraction performance is presented in Section 5.1, although it is also demonstrated implicitly in the ability of the system to provide the other four services. Section 5.2 describes the results of the performance analysis for the remaining four services.

### 5.1 Phase I: Validating the ARM Model

The model of the ARM architecture used by the front-end to translate ARM code to LLVM is validated using the LLVM interpreter (lli) and gdb. A set of ARM assembly test programs is hand-coded to exercise the instructions implemented in the model. These programs are executed in lli (cf. Figure 10) and in an ARM debugger and the results compared to verify the instructions were translated correctly.

```
Interpreting: %SP_16 = ptrtoint i8* %SP_13 to i32
--> i32 147585285 (0x8CBF905) (i32 100256520 byteswapped)

Interpreting: %SP_17 = sub i32 %SP_16, 1
--> i32 147585284 (0x8CBF904) (i32 83479304 byteswapped)

Interpreting: %SP_18 = inttoptr i32 %SP_17 to i8*
--> void* 147585284

Interpreting: %R2_ = add i32 2, 1
--> i32 3 (0x3) (i32 50331648 byteswapped)

Interpreting: %R1_25 = sub i32 %R2_, 1
--> i32 2 (0x2) (i32 33554432 byteswapped)
```

Figure 10. Executing a lifted LLVM program in the lli interpreter.

Table 3, Table 4 and Table 5 summarize the instructions validated in the prototype implementation of ABACAS. The current prototype implements 75% of branch instructions, 10% of data processing instructions and 8% of load/store and load/store multiple instructions.

**Table 3. Branch/call/return instructions validated.**

<b>Branch/Call/Return Instructions</b>	
Opcode	Condition Codes
B	AL, NE, LE, LT
BL	AL
BX	AL

**Table 4. Data-processing instructions validated.**

<b>Data Processing Instructions</b>		
Opcode	Operand Types	
	Imm	Reg
ADD	X	X
CMP	X	
MOV	X	X
SUB	X	X

**Table 5. Load/store and load/store multiple instructions validated.**

<b>Load/Store and Load/Store Multiple Instructions</b>										
Opcode	Operand Types		Addressing Modes							
	Imm	Reg	Offset	Pre-indexed	Post-indexed	IA	IB	DA	DB	Write-back
LDR	X		X	X	X	n/a	n/a	n/a	n/a	n/a
STR	X		X	X	X	n/a	n/a	n/a	n/a	n/a
LDM	n/a	n/a	n/a	n/a	n/a	X				X
STM	n/a	n/a	n/a	n/a	n/a				X	X

The ARM model is valid for all instructions marked with an X in Table 4 and Table 5, for all unconditional B, BL and BX instructions, and for BNE, BLE and BLT conditional branch instructions. Conditional execution of other instructions has not been validated.

## 5.2 Phase II: ABACAS Performance Analysis

### 5.2.1 Impact of source language on LLVM program analysis

A comparison of proportions determines if there is a statistically significant difference between analyses performed on LLVM programs compiled from C versus programs lifted from ARM machine code of the same programs. Two different analysis passes are executed on the programs, one to identify the number of loops in the programs and the second to identify dominance frontiers in the programs. Both the loop detection rate (LDR) and the dominance frontier detection rate (DDR) are 1, meaning both analyses identified all elements in programs translated from both source languages. There is no evidence of a difference in detection rates for either analysis.

Figure 11 and Figure 12 show the output from ABACAS for analysis of the `main` functions of a program which recursively calculates the first 10 numbers of the Fibonacci sequence.

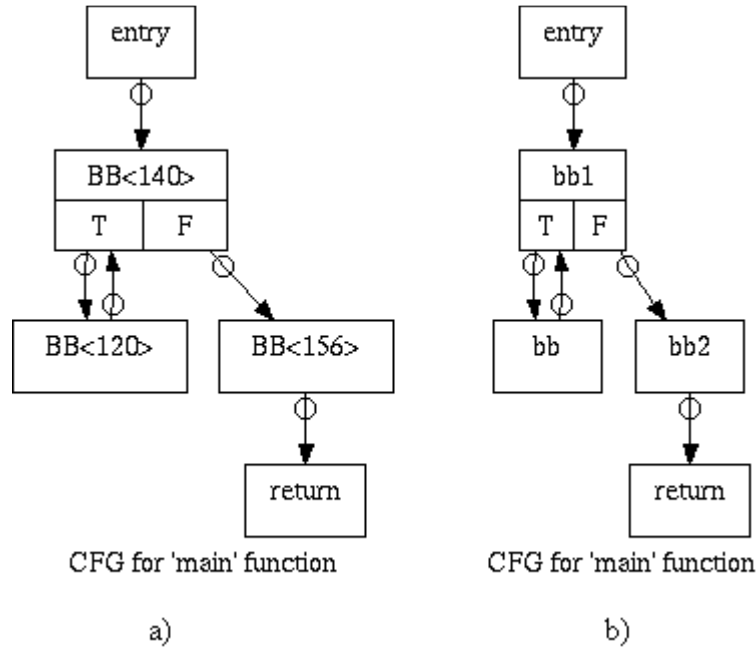
```
Printing analysis 'Natural Loop Information' for function 'main':
Loop at depth 1 containing: %"BB<140>"<header><exiting>,"BB<120>"<latch>
Printing analysis 'Dominance Frontier Construction' for function 'main':
DomFrontier for BB %entry is:
DomFrontier for BB %"BB<140>" is:      %"BB<140>"
DomFrontier for BB %"BB<120>" is:      %"BB<140>"
DomFrontier for BB %return is:
DomFrontier for BB %"BB<156>" is:
```

Figure 11. Loop and dominance frontier analyses on program lifted from ARM machine code.

```
Printing analysis 'Natural Loop Information' for function 'main':
Loop at depth 1 containing: %bb1<header><exiting>,%bb<latch>
Printing analysis 'Dominance Frontier Construction' for function 'main':
DomFrontier for BB %entry is:
DomFrontier for BB %bb1 is:      %bb1
DomFrontier for BB %bb is:      %bb1
DomFrontier for BB %bb2 is:
DomFrontier for BB %return is:
```

Figure 12. Loop and dominance frontier analyses on same program compiled from C.

Figure 13 shows the control flow graphs of these two functions. The loop identified in Figure 11 between basic blocks  $BB<140>$  and  $BB<120>$  is clearly visible in Figure 13a and corresponds to the loop identified in Figure 12 between blocks  $bb1$  and  $bb$  and visible in Figure 13b. The dominance frontiers identified in Figure 11 and Figure 12 include equivalent nodes  $BB<140>$  and  $bb1$  respectively.



**Figure 13. Control flow graphs of main function of the Fibonacci program a) lifted from ARM code and b) compiled from C.**

### 5.2.2 Impact of source language on size of generated LLVM assembly code.

Figure 14 and Figure 16 show two versions of the Fibonacci program translated to LLVM from two different source languages. The version in Figure 14 was compiled from the C source code shown in Figure 15 and the version in Figure 16 was lifted from ARM machine code. The ARM assembly which produced the ARM machine code is shown in Figure 17. The instruction count of LLVM IR lifted from machine code is

much greater for this program than LLVM IR generated from the C code. A similar result is observed in every sample program submitted to the system.

```
define i32 @main() nounwind {
entry:
    %retval = alloca i32
    %0 = alloca i32
    %i = alloca i32
    %value = alloca i32
    %"alloca point" = bitcast i32 0 to i32
    store i32 0, i32* %i, align 4
    br label %bb1

bb:
    %1 = load i32* %i, align 4
    %2 = call i32 @fib(i32 %1) nounwind
    store i32 %2, i32* %value, align 4
    %3 = load i32* %i, align 4
    %4 = add nsw i32 %3, 1
    store i32 %4, i32* %i, align 4
    br label %bb1

bb1:
    %5 = load i32* %i, align 4
    %6 = icmp sle i32 %5, 9
    br i1 %6, label %bb, label %bb2

bb2:
    %7 = load i32* %value, align 4
    store i32 %7, i32* %0, align 4
    %8 = load i32* %0, align 4
    store i32 %8, i32* %retval, align 4
    br label %return

return:
    %retval3 = load i32* %retval
    ret i32 %retval3
}
```

**Figure 14. main function of Fibonacci program compiled to LLVM IR from C source code**

```
int fib(int x) {
    if (!x)
        return 0;
    if (x > 2)
        return (fib(x - 1) + fib(x - 2));
    return 1;
}

int main() {
    int i, value;

    for (i = 0; i < 10; i++) {
        value = fib(i);
    }
    return value;
}
```

**Figure 15. C source code of Fibonacci program.**

```

define i32 @main() {
entry:
  %stack_vars = alloca [20 x i8]
  %SP_ = getelementptr inbounds [20 x i8]* %stack_vars, i32 0, i32 20
  %0 = ptrtoint i8* %SP_ to i32
  %1 = sub i32 %0, 4
  %SP_5 = inttoptr i32 %1 to i8*
  %2 = bitcast i8* %SP_5 to i32*
  store i32 undef, i32* %2
  %SP_8 = ptrtoint i8* %SP_5 to i32
  %SP_9 = sub i32 %SP_8, 16
  %SP_10 = inttoptr i32 %SP_9 to i8*
  br label %"BB<140>"

"BB<140>":
  %R0_11.0 = phi i32 [ 0, %entry ], [ %R0_50, %"BB<120>" ]
  %CPSR_.0 = phi i32 [ undef, %entry ], [ %CPSR_18, %"BB<120>" ]
  %3 = ptrtoint i8* %SP_10 to i32
  %4 = add i32 %3, 4
  %5 = inttoptr i32 %4 to i8*
  %6 = bitcast i8* %5 to i32*
  store i32 %R0_11.0, i32* %6
  %7 = ptrtoint i8* %SP_10 to i32
  %8 = add i32 %7, 4
  %9 = inttoptr i32 %8 to i32*
  %R0_15 = load i32* %9
  %CPSR_18 = call i32 @arm_cmp(i32 %CPSR_.0, i32 %R0_15, i32 10)
  %CPSR_V = and i32 %CPSR_18, 268435456
  %V = icmp eq i32 %CPSR_V, 268435456
  %CPSR_N = and i32 %CPSR_18, -2147483648
  %N = icmp eq i32 %CPSR_N, -2147483648
  %CPSR_LT = icmp ne i1 %N, %V
  br i1 %CPSR_LT, label %"BB<120>", label %"BB<156>"

"BB<156>":
  %10 = ptrtoint i8* %SP_10 to i32
  %11 = add i32 %10, 0
  %12 = inttoptr i32 %11 to i32*
  %R0_20 = load i32* %12
  %13 = ptrtoint i8* %SP_10 to i32
  %14 = add i32 %13, 12
  %15 = inttoptr i32 %14 to i8*
  %16 = bitcast i8* %15 to i32*
  store i32 %R0_20, i32* %16
  %17 = ptrtoint i8* %SP_10 to i32
  %18 = add i32 %17, 8
  %19 = inttoptr i32 %18 to i8*
  %20 = bitcast i8* %19 to i32*
  store i32 %R0_20, i32* %20
  %21 = ptrtoint i8* %SP_10 to i32
  %22 = add i32 %21, 12
  %23 = inttoptr i32 %22 to i32*
  %R0_26 = load i32* %23
  %SP_28 = ptrtoint i8* %SP_10 to i32
  %SP_29 = add i32 %SP_28, 16
  %SP_30 = inttoptr i32 %SP_29 to i8*
  %24 = bitcast i8* %SP_30 to i32*
  %LR_32 = load i32* %24
  %25 = ptrtoint i8* %SP_30 to i32
  %26 = add i32 %25, 4
  %SP_33 = inttoptr i32 %26 to i8*
  br label %return

"BB<120>":
  %27 = ptrtoint i8* %SP_10 to i32
  %28 = add i32 %27, 4
  %29 = inttoptr i32 %28 to i32*
  %R0_36 = load i32* %29
  %R0_44 = call i32 @"func<0>"(i32 %CPSR_18, i32 undef, i32 %R0_36, i32 undef)
  %30 = ptrtoint i8* %SP_10 to i32
  %31 = add i32 %30, 4
  %32 = inttoptr i32 %31 to i32*
  %R1_ = load i32* %32
  %33 = ptrtoint i8* %SP_10 to i32
  %34 = add i32 %33, 0
  %35 = inttoptr i32 %34 to i8*
  %36 = bitcast i8* %35 to i32*
  store i32 %R0_44, i32* %36
  %R0_50 = add i32 %R1_, 1
  br label %"BB<140>"

return:
  ret i32 %R0_26
}

```

**Figure 16.** main function of Fibonacci program lifted from ARM code.



```

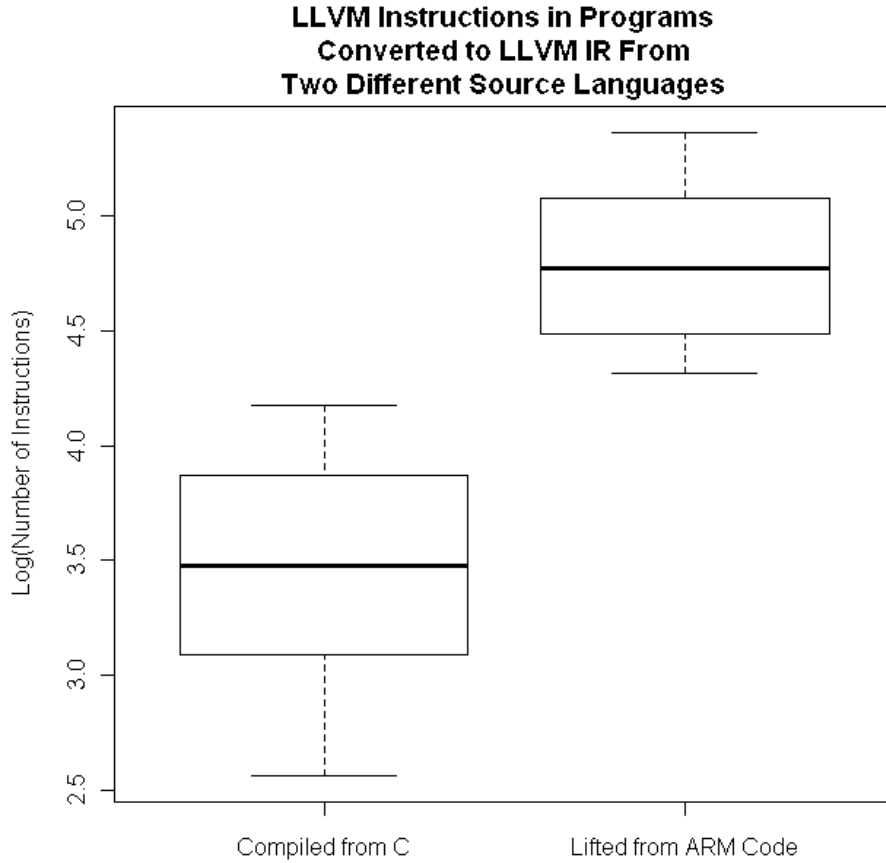
main:
@ BB#0:

    str    lr, [sp, #-4]!
    sub    sp, sp, #16
    mov    r0, #0
    b      .LBB1_2
.LBB1_1:
    ldr    r0, [sp, #4]
    bl     fib
    ldr    r1, [sp, #4]
    str    r0, [sp]
    add    r0, r1, #1
.LBB1_2:
    str    r0, [sp, #4]
    ldr    r0, [sp, #4]
    cmp    r0, #10
    blt    .LBB1_1
@ BB#3:
    ldr    r0, [sp]
    str    r0, [sp, #12]
    str    r0, [sp, #8]
    ldr    r0, [sp, #12]
    add    sp, sp, #16
    ldr    lr, [sp], #4
    bx     lr

```

**Figure 17. ARM assembly code of main function of Fibonacci program.**

Figure 18 is box plots of LLVM instruction count in programs compiled from C and programs lifted from ARM machine code. A two-sample t-test is used to determine if there is a statistically-significant difference between the two groups. With a one-sided p-value of  $9.90 \times 10^{-12}$  there is convincing evidence that programs lifted by ABACAS from ARM code are larger than those compiled from C source code. The difference in log-means is estimated to be 1.3165. Transforming back to an instruction count, the size of LLVM programs lifted from ARM code is estimated to be an average of 3.73 times greater than programs generated from C with a 95% confidence interval of (2.84, 4.90).



**Figure 18. LLVM instructions in programs generated from C source and ARM code.**

#### *5.2.2.1 Interpretation of results*

The results indicate that the difference in mean program size is caused by the source language from which the program is translated. This is likely due to the fact that the ARM machine code version of each program is much larger than the C source code of the same program. For example, the `main` function of the Fibonacci program only consists of four C statements (cf. Figure 15) whereas the ARM assembly code of the same function consists of 20 instructions (cf. Figure 17). It could also be that ABACAS is not only concerned with functional correctness of the generated code, but on preserving

architectural details present in the ARM code such as the layout of data in memory and use of specific registers like the CPSR, SP and LR to enable valid security analyses. This modeling of ARM architectural details adds additional overhead in size of the generated code.

Although causality can be inferred for the set of programs tested, the results cannot be inferred to a wider population of programs, since those tested were not randomly selected.

### 5.2.3 *Impact of source language on LLVM restructuring transformations*

To determine the effectiveness of transformations on LLVM programs lifted by ABACAS from ARM machine code versus transformations on code compiled from C source, the rates of instructions removed by the `-instcombine` transformation are compared. Table 6 shows the total number of instructions eliminated by the transformation for each group of programs.

**Table 6. Instructions eliminated by `-instcombine` transformation on programs compiled from C source code and those lifted from ARM machine code.**

	<b>Instructions Eliminated</b>		Totals	IRR
	Eliminated	Not Eliminated		
C Source Code	66	289	355	0.19
ARM Code	186	1062	1248	0.15
Totals	252	1351	1603	

There is no conclusive evidence that the instruction reduction rate (IRR) is greater for programs compiled from C code. The estimated difference of 0.037 has a chi-squared statistic of 2.8368 and a one-sided p-value of 0.0461. However, the 95% confidence interval for the difference in proportions is (-0.00816, 0.08191), which includes 0.

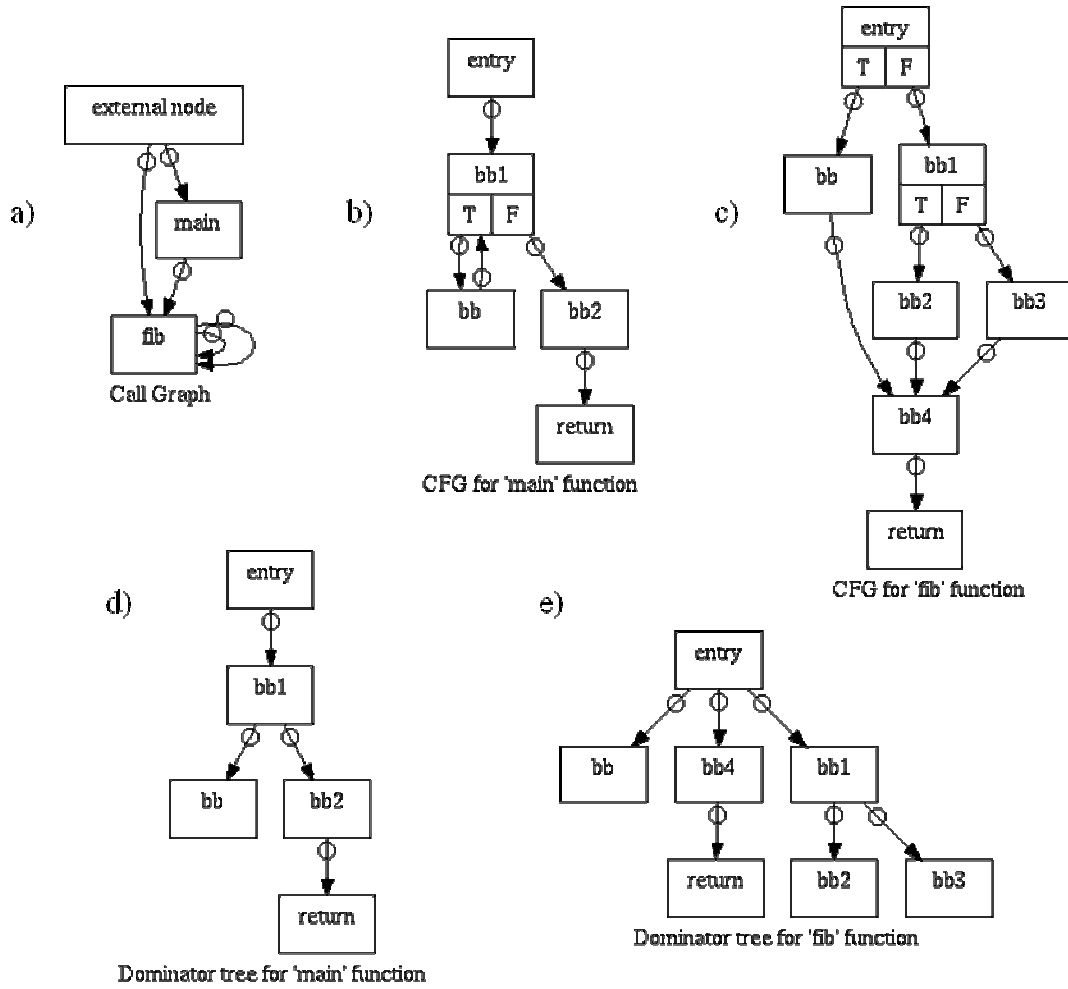
A Mantel-Haenszel test provides an estimate of the common odds ratio while accounting for the *control-flow constructs* factor. The estimated common odds ratio is 1.304 with a 95% confidence interval of (0.9577, 1.7759). Since the confidence interval includes 1, with only a moderately low p-value of 0.0450, there is still no conclusive evidence that the odds of eliminating instructions are higher for programs compiled from C even after accounting for the different control-flow constructs in the tested programs.

#### *5.2.3.1 Interpretation of results*

The results provide no evidence that the LLVM programs compiled from C have a higher IRR than those translated from ARM. However, the set of programs analyzed was not selected randomly so inference cannot be made to a wider population. Moreover, this one transformation is not representative of all possible instruction-reducing transformations available on LLVM IR. More tests are required to determine how other transformations perform on ARM machine code.

#### *5.2.4 Impact of source language on LLVM program redocumentation*

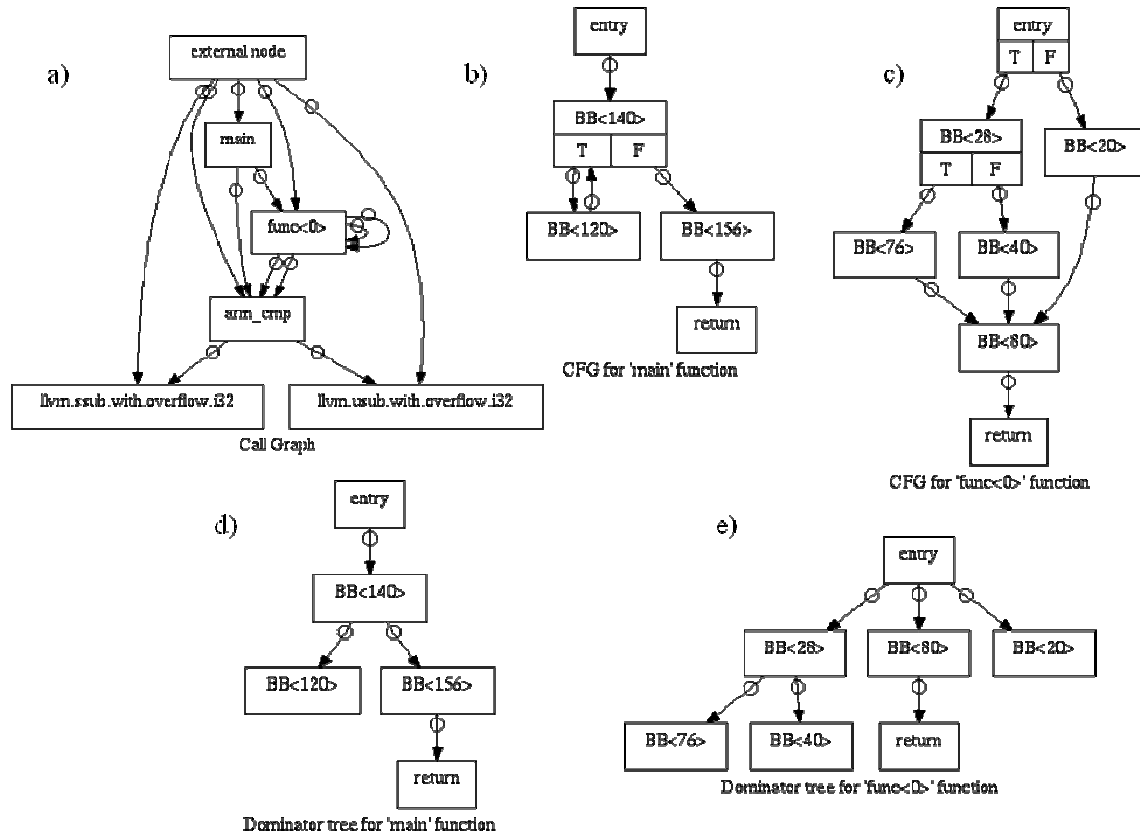
All views attempted were successfully displayed in programs translated from both source languages. These include a call graph, a control flow graph (CFG) for each function in the program and a dominator tree for each function in the program. As the view generation rate (VGR) is 1 for both groups, there is no evidence of a difference in proportions of views generated. Figure 19 and Figure 20 show these views generated from the Fibonacci program translated from C and from ARM.



**Figure 19. Views generated from the Fibonacci program compiled from C source code: a) the call graph, b) CFG for `main` function, c) CFG for `fib` function, d) dominator tree for `main` function, and e) dominator tree for `fib` function.**

Two apparent differences in the views require explanation. First, the call graph of the program lifted from ARM code includes several function calls that are not present in the call graph of the C-compiled program. This is a result of ABACAS' translation strategy for the ARM `CMP` instruction. The ARM architecture sets various flag bits in the current program status register (CPSR) according to the result of a `CMP` instruction. Subsequent conditional instructions execute if the relevant CPSR flag bits are set. ABACAS models this functionality by translating the `CMP` instruction as a separate

function call to the `arm_cmp` function. This function takes as parameters the CPSR register, and two registers to compare and returns the value of CPSR with the appropriate flag bits set. This approach correctly models the ARM architecture while reducing code size and improving readability of the generated code.



**Figure 20.** Views generated from the Fibonacci program lifted from ARM machine code: a) the call graph, b) CFG for `main` function, c) CFG for `fib` function, d) dominator tree for `main` function, and e) dominator tree for `fib` function.

Second, the true/false conditions appear to be reversed in the CFG for the `fib` functions for each program (cf. Figure 19c and Figure 20c). This is due to the way the LLVM back-end compiles LLVM IR to ARM assembly. The LLVM instructions in Figure 21a branch to `%bb2` if the value of register `%3` is greater than 2. The back-end compiles these to the ARM instructions in Figure 21b, which branch to the opposite

block if the register is less than 3. Although the conditions are reversed, the branch destinations are reversed as well so the programs are functionally equivalent. ABACAS generates code from the ARM instructions, thus the apparent difference in the CFGs.

<pre>%4 = icmp sgt i32 %3, 2 br i1 %4, label %bb2, label %bb3</pre>	<pre>cmp    r0, #3 blt    .LBB0_4</pre>
a)	b)

**Figure 21. Conditional branch compiled a) from C to LLVM, and b) from LLVM to ARM assembly.**

### 5.2.5 *Impact of source language on recompilation*

Another comparison of proportions determines whether source language of the input program significantly impacts the ability to compile the IR to multiple architectures. The LLVM programs, translated from C source code and ARM machine code are recompiled to two different binaries, one targeting the ARM architecture and one targeting the Intel x86 architecture. Both programs are executed in debuggers and the outputs analyzed to determine the recompilation success rates (RSRs). Figure 22 shows one of these programs being debugged in an x86 debugger and an ARM debugger.

The program recursively calculates the first 10 numbers in the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34. In Figure 22a, the x86 version of the program, register `$ecx` holds the iteration value (ranging from 0 to 9) and register `$eax` contains the Fibonacci number calculated for that iteration. In Figure 22b, register `$r1` holds the iteration value and register `$r0` holds the Fibonacci number for that iteration. Both programs return 34, the 10<sup>th</sup> Fibonacci number, via registers `$eax` and `$r0` respectively.

<pre> 0x804840e &lt;main+46&gt;:  mov    %eax,0x14(%esp) (gdb) c Continuing.  Breakpoint 2, 0x0804840e in main () 3: \$ecx = 6 2: \$eax = 8 1: x/i \$pc 0x804840e &lt;main+46&gt;:  mov    %eax,0x14(%esp) (gdb) c Continuing.  Breakpoint 2, 0x0804840e in main () 3: \$ecx = 7 2: \$eax = 13 1: x/i \$pc 0x804840e &lt;main+46&gt;:  mov    %eax,0x14(%esp) (gdb) c Continuing.  Breakpoint 2, 0x0804840e in main () 3: \$ecx = 8 2: \$eax = 21 1: x/i \$pc 0x804840e &lt;main+46&gt;:  mov    %eax,0x14(%esp) (gdb) █  Breakpoint 2, 0x0804840e in main () 3: \$ecx = 9 2: \$eax = 34 1: x/i \$pc 0x804840e &lt;main+46&gt;:  mov    %eax,0x14(%esp) (gdb) c Continuing.  Breakpoint 3, 0x08048453 in main () 3: \$ecx = 34 2: \$eax = 34 1: x/i \$pc 0x8048453 &lt;main+115&gt;:  ret (gdb) c Continuing.  Program exited with code 042. (gdb) </pre>	<pre> Breakpoint 2, 0x000083d4 in main () 3: \$r1 = 6 2: \$r0 = 8 1: x/i \$pc 0x83d4 &lt;main+32&gt;:      str    r0, [sp] (gdb) c Continuing.  Breakpoint 2, 0x000083d4 in main () 3: \$r1 = 7 2: \$r0 = 13 1: x/i \$pc 0x83d4 &lt;main+32&gt;:      str    r0, [sp] (gdb) _  Breakpoint 2, 0x000083d4 in main () 3: \$r1 = 8 2: \$r0 = 21 1: x/i \$pc 0x83d4 &lt;main+32&gt;:      str    r0, [sp] (gdb) c Continuing.  Breakpoint 2, 0x000083d4 in main () 3: \$r1 = 9 2: \$r0 = 34 1: x/i \$pc 0x83d4 &lt;main+32&gt;:      str    r0, [sp] (gdb) c Continuing.  Breakpoint 3, 0x00008418 in main () 3: \$r1 = 0 2: \$r0 = 34 1: x/i \$pc 0x8418 &lt;main+100&gt;:     pop    {r4, pc} (gdb) c Continuing.  Program exited with code 042. (gdb) _ </pre>
---	--

a)

b)

**Figure 22.** Debugging the Fibonacci program a) in an x86 debugger and b) in an ARM debugger.

All programs correctly executed on both ARM and x86 machines regardless of whether the input program was translated to LLVM from C source code or from ARM machine code. There is no evidence of a difference in RSRs for the programs tested.

### 5.3 *Summary of results*

The experimental results demonstrate that ABACAS successfully provides its system services, which include code abstraction, code analysis, code redocumentation,



code restructuring and code reengineering, for the submitted workload. Analysis passes performed on LLVM assembly code abstracted from ARM binary code produce the same loop detection and dominance frontier detection rates as analysis passes performed on LLVM assembly compiled from C source code. Equivalent views of the programs, including call graphs, control flow graphs and dominance trees, are demonstrated on LLVM code translated from the two different sources. One restructuring transformation applied to LLVM programs translated from the two different sources is successfully demonstrated on programs translated from both sources. Finally, the ability to reengineer a program is demonstrated by lifting several simple programs from ARM machine code to LLVM IR, performing restructuring transformations to reduce redundant instructions in the generated code, recompiling the transformed programs to a different machine architecture (Intel x86) and executing the programs in an x86 debugger.

## VI. Conclusions

### **6.1    *Research Accomplishments***

The ABACAS architecture is based on the Low-level Virtual Machine compiler framework and analyzes programs with a very wide range of abstraction. The prototype front-end designed and developed herein lifts binary executables compiled for the ARM architecture to the LLVM IR. More specifically, a machine code parser is implemented which uses a recursive-descent predictive parsing algorithm to produce an abstract syntax tree (AST) of an ARM executable. This parsing approach allows ABACAS to lift binaries devoid of symbol table, string table and debugging information. A code generator is implemented to translate a subset of ARM instructions into valid LLVM IR. The prototype currently supports 75% of ARM branch instructions, 10% of ARM data-processing instructions and 8% of load/store instructions.

By lifting binary executables to the LLVM intermediate representation, ABACAS exploits the program analysis, program transformation, code visualization and forward compilation capabilities of the LLVM open-source compiler framework. The experiments in Chapters 4 and 5 demonstrate a subset of these capabilities on programs lifted from ARM machine code, including two analysis passes, three different graphical views of the programs, one program transformation and recompilation of the programs to two target processor architectures, ARM and Intel x86.

## 6.2 *Contributions*

The primary contribution of this research is the translation of binary machine code directly into an architecture-independent compiler IR such that all typical compiler functions can be applied just as if the IR was compiled from a high-level-language. There have been attempts to use other compiler systems in this way as described in Section 2.2.4.3 but these do not offer the architecture-independence of LLVM and levy impractical requirements on the binary itself. ABACAS does not require the binary to be compiled using any particular compiler or to have any special symbol information present.

Other contributions of this research derive from the ability to translate binary machine code into LLVM IR. This research makes nearly all existing and future LLVM analysis and transformation passes available for ARM binary programs that can be lifted by ABACAS. This is also one of the few successful static binary-to-binary translators developed.

## 6.3 *Applications of this Research*

### 6.3.1 *Automatic vulnerability discovery*

ABACAS could be used to detect vulnerabilities in binary programs. The LLVM pass framework makes it easy to leverage existing analyses and to write new ones by chaining analysis passes together. The SAFECODE project [DKA06] includes passes that may be useful for detecting possible memory corruption errors. The KLEE symbolic execution engine [CDE08] could traverse as many execution paths through the code as

possible, test for program bugs and automatically generate test cases which exercise the discovered errors to determine if they are true vulnerabilities.

#### *6.3.1 Improved malware analysis*

Automated binary abstraction and analysis services provided by ABACAS may be helpful in reverse-engineering and analyzing malware. When a new worm is released in the “wild” which exploits a previously unknown software bug, the malware must be quickly reverse-engineered to identify the vulnerability, then a software patch must be developed that corrects the vulnerability without introducing new vulnerabilities into the software. ABACAS’ code abstraction, code analysis and code reengineering services are well suited to these tasks.

#### *6.3.2 Software Maintenance*

ABACAS offers a level of flexibility in software development that would be very useful later in the software engineering life cycle. New functionality could be added to an existing executable program by programming the functionality in a high-level language, compiling it to LLVM, lifting the existing native machine code to LLVM, modify the LLVM code of the original program to call the new functions, link the LLVM files and compile back to native code. Software maintenance costs are estimated to be 50% to 90% of the total lifecycle costs of software [CC90]. Tools such ABACAS which help automate reverse-engineering, analyzing and modifying code could be crucial in reducing these costs.

#### *6.3.2.1 Binary rewriting for improved code security*

ABACAS could be used to modify a binary program to eliminate detected vulnerabilities. After lifting the binary to LLVM, modifications could be made directly in LLVM and the program re-tested. After the entire program is “hardened,” it could be recompiled back to native code.

#### *6.3.2.2 Port software to new machine architectures without source code*

The code reengineering experiment described in Chapters 4 and 5 demonstrated the ability to lift a program compiled for one architecture and recompile it to another machine architecture. It may be possible to do this on a larger scale for other programs or libraries when source code is not available.

### **6.4 Future Work**

#### *6.4.1 Expand object file support*

The ABACAS prototype relies on IDA Pro to retrieve the hexadecimal program bytes required by the parser and disassembler and to resolve all relocation information. The current approach has several disadvantages: 1) relying on IDA Pro, an external, commercial program, prevents ABACAS from being a self-contained system and adds unnecessary steps to manually retrieve the program information before feeding it into the parser. This significantly slows down the process of analyzing binaries and would greatly benefit from a native object file parsing capability within ABACAS, 2) ABACAS does not currently use any imported function information from the object file. This means only self-contained programs can be lifted. Virtually every real-world program uses imported libraries so this is an essential capability, 3) although it is good that

ABACAS does not depend on symbol table, string table or debugging information, this information could vastly improve the quality and readability of lifted code and should be incorporated as supplementary information.

#### *6.4.2 Expand support for ARM*

One obvious necessity in the development of ABACAS is to expand support for the ARM architecture. Only a very small subset of instructions is currently implemented, severely limiting the programs which can be analyzed. Support for each instruction is manually coded in an ad-hoc fashion in the prototype ARM front-end. A more rigorous approach would be to capture all the features of a machine architecture required for translation in a description file and utilize LLVM's TableGen framework [Lat10] to auto-generate lookup tables to facilitate translation from the architecture-specific instructions to LLVM code.

#### *6.4.3 Add support for other architectures*

Adding support for other machine architectures would require little effort in the parser, but significant work in the code generator. Again, utilizing TableGen would help condense architecture-specific code primarily to the description file for each architecture. Intel's x86 architecture is an obvious next choice for an ABACAS front-end.

#### *6.4.4 Incorporate LLVM passes to improve system services*

##### *6.4.4.1 Improve re-compilation through a stack lowering pass*

All memory references that are translated to references in the modeled stack (the `%stack_vars` array) could be converted to allocas, loads and stores to the LLVM stack

frame for the function. This would eliminate unnecessary overhead and make powerful transformations available that operate on `alloca` instructions (such as the `mem2reg` pass).

#### *6.4.4.2 Improve decompilation to LLVM IR through iteration and analysis*

One of the problems that plagues recursive-descent disassembly is the inability to handle indirect branch addresses. LLVM analysis passes, including alias analysis and constant propagation, could be run after the first translation pass to attempt to resolve as many indirect addresses as possible before performing an additional translation pass with the new information. A third translation pass could even be implemented which catches missed code through iterative disassembly of the text segment in the object file.

#### *6.4.4.3 Improve program analysis by writing security-related passes*

Analysis passes should be written to enable vulnerability discovery in binary programs. These could include memory access checks such as array bounds checking, propagation of taint information from user inputs and marking certain program inputs as symbolic to enable symbolic execution of the program [CDE08].

## **6.5 Conclusion**

A modern optimizing compiler framework like LLVM, which provides a platform-independent IR, is a promising framework for architecture-independent program analysis, transformation and recompilation of binary programs. This research presents the design, implementation and demonstration of one such system, the Architecture-independent Binary Abstracting Code Analysis System (ABACAS). Although the prototype has many limitations and only implements a small subset of ARM instructions,

the results are still profound. Existing analysis, transformation, redocumentation and compilation capabilities are applied directly to programs lifted from binary format. These capabilities are crucial for protecting, hardening and attacking mobile devices and other modern systems which operate in the Cyber warfighting domain.



## Bibliography

- [AE02] K. Ashcraft and D. Engler, "Using programmer-written compiler extensions to catch security holes," in *Proceedings of the 2002 IEEE symposium on security and privacy*, Berkeley, CA, 2002, pp. 143-159.
- [And94] Lars Ole Andersen, "Program analysis and specialization for the C programming language," DIKU, University of Copenhagen, Copenhagen, Denmark, Ph.D. Thesis 1994.
- [ARM08] ARM, *ARM Architecture Reference Manual ARMv7-A and ARMv7-R*: ARM Ltd, 2004-2008.
- [ARM09] ARM Limited, "Procedure call standard for the ARM architecture," ARM Limited, ARM ABI release 2.08 2009.
- [ARM10] ARM. "Annual Report and Accounts 2009." (2010) ARM.com. [Online]. <http://www.arm.com/annualreport09/overview/industry-dynamics.html>
- [ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: principles, techniques, and tools*, James T. DeWolf, Ed. Reading, MA, USA: Addison-Wesley, 1988.
- [BA08] Domagoj Babic and Alan J. Hu, "Calysto: Scalable and Precise Extended Static Checking," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS'08)*, Leipzig, Germany, Mar. 2008, pp. 211-220.
- [BFL07] Michael Becher, Felix C. Freiling, and Boris Leider, "On the effort to create smartphone worms in windows mobile," in *Proceedings from the 8th annual IEEE SMC Information Assurance Workshop, IAW '07*, USMA, West Point, NY, 2007.
- [Bis96] Matt Bishop, "Checking for race conditions in file accesses," *Computing Systems*, vol. 9, no. 2, pp. 131-152, Spring 1996.
- [Bis03] Matt Bishop, *Computer security: art and science*. Boston: Addison-Wesley, 2003.
- [BJ10] David Brumley and Ivan Jager. (2009, May) The BAP Handbook. [Online]. <http://bap.ece.cmu.edu/doc/bap.pdf>
- [BR04] Gogul Balakrishnan and Thomas Reps, "Analyzing memory accesses in x86 executables," in *Proceedings of the 13th international conference on compiler construction*, Barcelona, Spain, 2004, pp. 5-23.

- [BR10] Gogul Balakrishnan and Thomas Reps, “WYSINWYX: What you see is not what you eXecute,” *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 6, August 2010.
- [CC90] Elliot J. Chikofsky and James H. Cross II, “Reverse Engineering and Design Recovery: A Taxonomy,” *IEEE Software*, vol. 7, no. 1, pp. 13-17, Jan./Feb. 1990.
- [CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, San Diego, CA, December 2008.
- [CFR91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451-490, Oct. 1991.
- [CNN09] CNN. “Obama to get spy-proof smartphone.” (2009, January) CNN Politics.com. [Online]. <http://www.cnn.com/2009/POLITICS/01/22/obama.blackberry/index.html>
- [CPG04] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum, “Understanding data lifetime via whole system simulation,” in *Proceedings of the 13th USENIX security symposium*, San Diego, CA, 2004, pp. 22-22.
- [Cov10] Coverity. (2010) Coverity Static Analysis. [Online]. <http://www.coverity.com/products/static-analysis.html>
- [DKK07] Michael Dalton, Hari Kannan, and Christos Kozyrakis, “Raksha: A flexible information flow architecture for software security,” in *Proceedings of the 34th annual international symposium on computer architecture, ISCA '07*, New York, NY, 2007, pp. 482-493.
- [DA06] Dinakar Dhurjati and Vikram Adve, “Backwards-Compatible Array Bounds Checking for C with Very Low Overhead,” in *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, Shanghai, China, May 2006.
- [DKA05] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner, “Memory Safety Without Garbage Collection for Embedded Applications,” *ACM Transactions in Embedded Computing Systems (TECS)*, vol. 4, no. 1, pp. 73-111, Feb 2005.

- [DKA06] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve, "SAFECode: Enforcing Alias Analysis for Weakly Typed Languages," in *Proceedings of the 2006 ACM SIGPLAN Conference on Software Engineering (ICSE '06)*, Shanghai, China, May 2006, pp. 144-157.
- [Eag08] Chris Eagle, *The IDA pro book: the unofficial guide to the world's most popular disassembler*, Megan Dunchak, Ed. San Francisco, U.S.A.: No Starch Press, 2008.
- [FH08] Jon Friedman and Daniel V. Hoffman, "Protecting data on mobile devices: a taxonomy of security threats to mobile computing and review of applicable defenses," *Information Knowledge Systems Management*, vol. 7, no. 1/2, pp. 159-180, 2008.
- [FSe09] F-Secure. (2009) Trojan: SymbOS/Skulls.A. [Online]. <http://www.f-secure.com/v-descs/skulls.shtml>
- [Gar10] Gartner. "Gartner Says Worldwide Mobile Phone Sales Grew 17 Per Cent in First Quarter 2010." (2010, May) Gartner. [Online]. <http://www.gartner.com/it/page.jsp?id=1372013>
- [Gra10] Grammatech. (2010) CodeSonar. [Online]. <http://www.grammatech.com/products/codesonar/overview.html>
- [HBC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi, "Interprocedural pointer alias analysis," *ACM transactions on programming languages and systems*, vol. 21, no. 4, pp. 848-894, July 1999.
- [Hex08] Hex-Rays SA. (2008) Debugging Symbian applications with IDA Pro. [Online]. [http://www.hex-rays.com/idapro/debugger/symbian\\_primer.pdf](http://www.hex-rays.com/idapro/debugger/symbian_primer.pdf)
- [HJO08] Sheikh Mahbub Habib, Cyril Jacob, and Tomas Olovsson, "A Practical Analysis of the Robustness and Stability of the Network Stack in Smartphones," in *Proceedings of 11th International Conference on Computer and Information Technology (ICCIT 2008)*, Khulna, Bangladesh, 2008, pp. 393-398.
- [HLV05] Michael Howard, David LeBlanc, and John Viega, *19 Deadly Sins of Software Security Programming Flaws and How to Fix Them.*: McGraw-Hill Osborne Media, 2005.
- [HM04] Greg Hoglund and Gary McGraw, *Exploiting software: how to break code*. Boston, Massachusetts, U.S.A.: Addison-Wesley, 2004.
- [Hof07] Daniel V. Hoffman, *Blackjacking: security threats to blackberry devices, PDAs and cell phones in the enterprise*. Indianapolis, Indiana, U.S.A.: Wiley Publishing, 2007.

- [Kas03] Kris Kaspersky, *Hacker disassembling uncovered*, Natalia Tarkova, Ed. Wayne, PA, U.S.A.: A-List, 2003.
- [Kas10] Martin Kaste. “Wipeout: When Your Company Kills Your iPhone.” (2010, November) NPR.org. [Online]. <http://www.npr.org/2010/11/22/131511381/wipeout-when-your-company-kills-your-iphone>
- [KBA02] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe, “Secure execution via program shepherding,” in *Proceedings of the 11th USENIX Security Symposium*, Berkeley, CA, 2002, pp. 191-206.
- [Kei09] Gregg Keizer. “iPhone owners demand to see Apple source code.” (2009, November) Computerworld. [Online]. [http://www.computerworld.com/s/article/9141222/iPhone\\_owners\\_demand\\_to\\_see\\_Apple\\_source\\_code](http://www.computerworld.com/s/article/9141222/iPhone_owners_demand_to_see_Apple_source_code)
- [KLA04] Jack Koziol et al., *The Shellcoder's Handbook*. Indianapolis, Indiana, U.S.A.: Wiley, 2004.
- [LA04] Chris Lattner and Vikram Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO-04)*, Palo Alto, CA, Mar. 2004.
- [Lan92] William Landi, “Undecidability of Static Analysis,” *ACM letters on programming languages and systems*, vol. 1, no. 4, pp. 323-337, December 1992.
- [Lat10] Chris Lattner. (2010, July) The LLVM Compiler Infrastructure. [Online]. <http://llvm.org/>
- [Latt10] Chris Lattner. (2010, April) Intro to the LLVM MC project. [Online]. <http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html#more>
- [Lat11] Chris Lattner. (2011, January) The LLVM target-independent code generator. [Online]. <http://llvm.org/docs/CodeGenerator.html#codegendesc>
- [LE01] David Larochelle and David Evans, “Statically detecting likely buffer overflow vulnerabilities,” in *Proceedings of the 10th USENIX security symposium*, Washington, D.C., 2001, pp. 14-14.

- [LML08] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley, "Securing web applications with static and dynamic information flow tracking," in *Proceedings of the 2008 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, San Francisco, CA, 2008, pp. 3-12.
- [Mat10] MathWorks. (2010) PolySpace Embedded Software Verification. [Online]. <http://www.mathworks.com/products/polyspace/>
- [McG04] Gary McGraw, "Software Security," *IEEE Security and Privacy*, vol. 2, no. 2, pp. 80-83, March/April 2004.
- [ME07] Stephen McCamant and Michael D. Ernst, "A simulation-based proof technique for dynamic information flow," in *Proceedings of the 2007 workshop on programming languages and analysis for security, PLAS '07*, San Diego, California, 2007, pp. 41
- [MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32-44, Dec. 1990.
- [Mic11] Microsoft. (2011) Phoenix Connect. [Online]. <https://connect.microsoft.com/Phoenix>
- [Mil10] Capt. Christine D. Millette. "Air Force to implement handheld device changes." (2010, March) Air Force Space Command. [Online]. <http://www.afspc.af.mil/news/story.asp?id=123195273>
- [Mit11] MITRE Corp. Common vulnerabilities and exposures. [Online]. <http://cve.mitre.org/>
- [Mitr11] MITRE Corp. Common Weakness Enumeration. [Online]. <http://cwe.mitre.org/>
- [MM00] Gary McGraw and Greg Morrisett, "Attacking malicious code: a report to the infosec research council," *IEEE Software*, vol. 17, no. 5, pp. 33-41, September/October 2000.
- [MM09] Collin Mulliner and Charlie Miller, "Injecting SMS Messages into Smart Phones for Security Analysis," in *Proceedings of the 3rd USENIX conference on offensive technologies, WOOT '09*, Montreal, Canada, 2009, pp. 5-5.
- [MN08] Manuel Mendonca and Nuno Neves, "Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities," in *Proceedings of the 2008 Seventh European Dependable Computing Conference, EDCC-7 '08*, Washington, DC, 2008, pp. 110-119.
- [Mul08] Collin Mulliner, "Exploiting Symbian: Symbian exploitation and shellcode development (presentation)," in *Proceedings of the 25th Chaos Communication Congress (25C3)*, Berlin, Germany, 2008.

- [MV06] Collin Mulliner and Giovanni Vigna, "Vulnerability Analysis of MMS User Agents," in *Proceedings of the 22nd Annual Computer Security Applications Conference, ACSAC '06*, Miami Beach, Florida, 2006, pp. 77-88.
- [Myr09] Magnus O. Myreen, "Formal verification of machine-code programs," University of Cambridge, Cambridge, United Kingdom, PhD Dissertation ISSN 1476-2986, 2009.
- [NS07] Nicholas Nethercote and Julian Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *Proceedings of the SIGPLAN conference on programming language design and implementation*, vol. 42, no. 6, pp. 89-100, June 2007.
- [NS05] James Newsome and Dawn Song, "Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software," in *Proceedings of the 12th annual network and distributed systems security symposium*, San Diego, CA, 2005.
- [PM04] Bruce Potter and Gary McGraw, "Software security testing," *IEEE Security and Privacy*, vol. 2, no. 5, pp. 81-85, September/October 2004.
- [QP09] Daniel Quinlan and Thomas Panas, "Source code and binary analysis of software defects," in *Proceedings of the 5th annual workshop on cyber security and information intelligence research*, Oak Ridge, Tennessee, 2009, pp. 40:1-40:4.
- [Qui11] Dan Quinlan. (2011, January) ROSE. [Online]. <http://rosecompiler.org/>
- [San05] San, "Hacking Windows CE," *Phrack Magazine*, vol. 11, no. 63, Phile #0x06 of 0x14, July 2005.
- [SBY08] Dawn Song et al., "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th international conference on information systems security*, Berlin, Heidelberg, 2008, pp. 1-25.
- [SH10] Reid Spencer and Gordon Henriksen. "LLVM's Analysis and Transform Passes." (2010, October) LLVM.org. [Online]. <http://llvm.org/docs/Passes.html>
- [SLZ04] Edward G. Suh, Jaewook Lee, David Zhang, and Srinivas Devadas, "Secure program execution via dynamic information flow tracking," *SIGARCH Computer Architecture News*, vol. 32, no. 5, pp. 85-96, October 2004.
- [Spe10] Michael Spencer. (2010, November) "Object files in LLVM." Presentation at the 2010 LLVM developers' meeting. [Online]. <http://llvm.org/devmtg/2010-11/Spencer-ObjectFiles.pdf>

- [Ste91] Daniel F. Sterne, "On the buzzword "security policy"," in *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, Oakland, California, 1991, p. 219.
- [STF01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers," in *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., 2001, pp. 16-16.
- [TCM05] Katrina Tsipenyuk, Brian Chess, and Gary McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors," *IEEE Security and Privacy*, vol. 3, no. 6, pp. 81-84, November/December 2005.
- [Tho84] Ken Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, Aug 1984.
- [Val10] Valgrind Developers. (2010) Valgrind. [Online]. <http://valgrind.org/>
- [VM04] Denis Verdon and Gary McGraw, "Risk analysis in software design," *IEEE Security and Privacy*, vol. 2, no. 4, pp. 79-84, July/August 2004.
- [Web10] Web Application Security Consortium. WASC Threat Classification. [Online]. <http://projects.webappsec.org/Threat-Classification>
- [WFB00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proceedings of the network and distributed system security symposium, NDSS '00*, San Diego, CA, 2000, pp. 3-17.
- [Win03] Jeannette M. Wing, "A call to action: look beyond the horizon," *IEEE Security and Privacy*, vol. 1, no. 6, pp. 62-67, November/December 2003.
- [WM95] Reinhard Wilhelm and Dieter Maurer, *Compiler Design*, Stephen S. Wilson, Ed. Harlow, England: Addison-Wesley, 1995.
- [Zel09] Jeff Zeleny. "Obama Digs In for His BlackBerry." (2009, January) The New York Times. [Online]. <http://www.nytimes.com/2009/01/08/us/politics/08berry.html>

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 074-0188	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p><b>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</b></p>					
1. REPORT DATE (DD-MM-YYYY) 24-03-2011		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From – To) Sep 2009 – Mar 2011	
4. TITLE AND SUBTITLE  Automated Analysis of ARM Binaries using the Low-Level Virtual Machine Compiler Framework				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)  Jeffrey B. Scott, Capt, USAF				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)  Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER  AFIT/GCO/ENG/11-14	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)  INTENTIONALLY LEFT BLANK				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT  APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT  Binary program analysis is a critical capability for offensive and defensive operations in Cyberspace. However, many current techniques are ineffective or time-consuming and few tools can analyze code compiled for embedded processors such as those used in network interface cards, control systems and mobile phones.  This research designs and implements a binary analysis system, called the Architecture-independent Binary Abstracting Code Analysis System (ABACAS), which reverses the normal program compilation process, lifting binary machine code to the Low-Level Virtual Machine (LLVM) compiler's intermediate representation, thereby enabling existing security-related analyses to be applied to binary programs. The prototype targets ARM binaries but can be extended to support other architectures. Several programs are translated from ARM binaries and analyzed with existing analysis tools. Programs lifted from ARM binaries are an average of 3.73 times larger than the same programs compiled from a high-level language (HLL). Analysis results are equivalent regardless of whether the HLL source or ARM binary version of the program is submitted to the system, confirming the hypothesis that LLVM is effective for binary analysis.					
15. SUBJECT TERMS Binary analysis, reverse engineering, program analysis, vulnerability discovery, static analysis, cell phone security, mobile device security					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT  UU	18. NUMBER OF PAGES 103	19a. NAME OF RESPONSIBLE PERSON Dr. Rusty O. Baldwin, ENG
REPORT U	ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) (937) 255-6565 x4445; Rusty.Baldwin@afit.edu

**Standard Form 298 (Rev. 8-98)**

Prescribed by ANSI Std. Z39-18