Can You Give Me a Lift?

William Mahoney[1] and Adam Spanier[2]
[1]School of Interdisciplinary Informatics, University of Nebraska at Omaha, Nebraska, USA
[2]Cyber Systems Department, University of Nebraska at Kearney, Nebraska, USA

wmahoney@unomaha.edu
aspanier@unomaha.edu

Abstract: Static Analysis (SA) is the practice of examining computer program source code for errors or vulnerabilities outside the compiler's capabilities. To carry out Static Analysis on computer programs, various tools exist that parse and examine each line for known issues. These tools do not compile the program, nor do they run the program. Instead, they analyse the source code directly and infer properties about the program without executing it - thus "static" analysis. Static analysis is not new. Early UNIX contained a program called "lint" for static analysis; a tool that has now existed since the late 1970s (Johnson 1978). Increasingly, however, modern static analysis practices indicate a more focused intent; find cybersecurity weaknesses. At the 22nd European Conference on Cyber Warfare and Security, the authors presented the background of Intermediate Representations (IRs) and described how this "middleware" representation can be utilized for static analysis of source code for cybersecurity weaknesses. By examining an IR, potential flaws in the source code can be located. When utilizing the IR as opposed to the original high-level language, the static analysis process becomes independent of the original source language; if several languages such as C, Rust, and others all compile into the same IR, static analysis of the IR allows the analysis process to no longer be tied to the high-level language grammar or syntax. The previous paper implemented a literature survey of available IR analysis tools to discover prior work; the authors have subsequently advanced the research and are actively using an IR framework, LLVM, for vulnerability analysis. In this research, source code in a high-level language is first compiled to LLVM and the resulting IR is used for analysis. This approach uses a "code-to-IR" SA analysis preparation paradigm. At the same time, there is the potential for binary "lifters" to be used. These tools "lift" an executable program – binary machine instructions – back to LLVM. In this way, the paradigm can also be reversed such that static analysis of LLVM can be performed on source code compiled into LLVM, *or* on executable programs in the field that are "lifted". This begs: how effective are these "lifters"? In this work, the authors present experiences in installation and operation of several binary "lifters" available as open-source projects. Some are supported better than others, some operate better than others, and some don't operate at all. Those that do lead to the follow-up: Is the "lifted" code suitable for static analysis, or is it too obfuscated relative to the original program? This paper describes just that – our efforts and results in locating a binary "lifter" suitable for bringing executable program test cases back to LLVM for analysis by the cybersecurity vulnerability tool concurrently under development.

Keywords: Static analysis, Intermediate representations, Cybersecurity, Vulnerability detection

1. Introduction

Static software analysis (SA) refers to the evaluation of programs for flaws contained within each program's source code (Thompson 2021). Of the two types of analysis, "static" and "dynamic", the latter depends on the execution of the program, while the former examines the code at the source level but does not actually run the program. Generally, static analysis tools operate upon representations of the program by examining the high-level language used to author the software; a program written in C++ would use a static analysis tool specifically for C++, while a program written in Java would necessitate a Java-based analysis tool. Static analysis is not new; UNIX programs used "lint", a small program named after the small bits of fibre in a clothes dryer, as early as 1978 (Johnson 1978, 1979). More recent tools with similar names are still in use today (PC-lint 2023).

The authors have been investigating a novel static analysis approach that implements analysis procedures at the Intermediate Representation (IR) level associated with the compiler rather than at the source code level. For example, the popular Linux compiler suite GCC uses an IR named Gimple (GCC 2023), while the compiler tools based on CLANG use an IR named LLVM (LLVM 2023). As an aside, Gimple is a generic form of "simple" which was a project at McGill University, while LLVM was developed at the University of Illinois and was previously an acronym but is now used stand-alone. The advantage of using the compiler IR as opposed to the high-level code may already be apparent to the reader; if many high-level languages compile into the same IR, analysis at the IR level makes the SA tool independent of the original source code language; what the researchers call *Programming Language Agnostic.* For instance, many languages compile into LLVM, including Fortran, Haskell, Julia, Kotlin, Lua, Objective-C, OpenGL, Rust, and Swift, as well as other oddities including Pony, Crack, and others (LLVM 2023, Pony 2023, Crack 2023). Searching for certain software issues at the LLVM level thus covers all languages that compile into LLVM.

In the past, the authors have reported on research utilizing the LLVM IR in several ICCWS and ECCWS conferences (Mahoney 2021, Mahoney 2022, Spanier 2023). This prior research was aimed primarily at software intellectual

property protection, by “fingerprinting” individual copies of executable programs. The focus in this research has shifted; currently the aim of this project is two-fold. First, the authors are developing a domain-specific language (DSL) with the aim of taking an English software-flaw description and encoding it in an easy, readable language. The resulting description will be used to generate checking code which will examine LLVM for the specified flaw. The English-based descriptions chosen for this research come from the popular Common Weakness Enumeration (CWE) (Mitre 2023). The test cases which will be used for this effort come from the Software Assurance Reference Dataset (SARD) available from the National Institute of Standards and Technology (NIST) (SARD 2023). This research is ongoing, and the authors anticipate reporting on this effort in future publications. A second aim of the ongoing work is to use binary “lifters” to increase the potential coverage for the static analysis tool currently under development. This second aim is the topic of this paper.

What is a “binary lifter”? These are efforts to take an executable program – in binary form as a Windows EXE or a Linux ELF file – and convert the code to LLVM. LLVM, as an IR, is a higher-level representation than the actual program instructions, thus the term “lift”. The process of compiling has a similarity with the process of hashing: during the compiling process, information is lost. Due to this unavoidable loss of information, the process of lifting machine code back to the original high-level representation never results in a high-fidelity representation. Just as there are static and dynamic analysis tools, not executing versus executing the binary, there are static versus dynamic lifters. For this work, either method is acceptable, as the desire is to analyse test cases from SARD regardless of how they are lifted.

Lifting a binary executable back to an IR presents a possibility for this research: one can conceivably start from a Linux/Windows executable file, lift the executable to LLVM, and run static analysis tools on the result. In this way a program can be analysed for potential vulnerabilities, using the CWE data set, whether the original program was compiled *into* LLVM or whether the executable program (with no source code) was *lifted* to LLVM. This possibility theoretically provides a uniform SA tool for all compiled binaries of a given compiler.

The authors thus decided to “test drive” four binary lifting projects and report the results here. Of course, not all are created equal. Since they are open source, some have little to no documentation, some work “right out of the box” and others have serious flaws. This paper reports on the results of downloading, installing, and testing out four open-source projects for binary lifters. As a ramification, this paper presents little in terms of background and prior work, as the prior work is directly represented in the open-source projects described herein. The authors do report individually on the experiences encountered, and for those tools that operate, briefly refer the reader to a paper on performance information. A result is that the paper is organized differently than a traditional research paper. Sections two through five present each of the four lifters, in turn, and overall conclusions are presented in section six.

2. A Failure: McSema

The authors started from a well-known lift tool named McSema, which was presented initially at the Montreal Recon Conference (McSema1 2023, Recon 2014). The McSema team also make available a quite good video of their presentation via YouTube (McSema2 2023) and the source code is in an open repository at GitHub. Hopes are high. From the repository:

> *“McSema is an executable lifter. It translates ("lifts") executable binaries from native machine code to LLVM bitcode. LLVM bitcode is an intermediate representation form of a program that was originally created for the retargetable LLVM compiler, but which is also very useful for performing program analysis methods that would not be possible to perform on an executable binary directly.” (McSema1 2023)*

Although the project is dated, the repository that is openly available does have modifications dated 2021; the authors assume that it is “current enough” that it can be made to work, although possibly requiring Linux and other versions of software from that time period.

But the installation of McSema is, like many open-source projects, fraught with issues. There are two primary problems installing and using this tool. First, the documentation is scant with respect to the necessary required prerequisite packages; some of them are listed in the documentation, but many are not, and apparently the unlisted packages are assumed to be necessary for a correct build of the tool. The documentation for McSema also states that it relies on several other tools which need to be installed first, specifically “remill” and “anvill”. Unfortunately, these projects have similar issues, for example the latter tool requires the package “doctest”, which is not listed as needed for McSema. So, the authors here fall into a loop: find what the build process says

is missing, locate which package for Linux includes that feature, install that package using the Linux installer, start again, discovering the next package. Repeat.

A second problem with McSema is that the documentation that exists on the repository is outdated. For example, the install instructions give very specific releases for the necessary tools, like calling out "release_93aba7c" for "remill", when no such release exists, and showing an example build using LLVM 11 when the only version that seems to make forward progress is version 12, and the current stable version is 15. Additional permissions are needed to copy the files into the correct destination directories, which is a minor issue but is not mentioned in the documentation.

Lastly, McSema looks for a package named "xed" (intel 2023). This is a collection of C code, supplied by intel, which is used to decode x86-64 instructions in software. It is easily downloaded and compiled, and the authors have used "xed" several times with no issues on prior research projects, e.g. (Mahoney 2023). It is not, however, a Linux "package", so despite downloading and compiling "xed", McSema cannot find it as a package, and building the tool stops.

The first attempt at a working binary lifter, McSema, is a failure. However, all is not lost, as the documentation for McSema includes a comparison to other lifter packages, along with links. If McSema is problematic, at least they point in a new direction. Time to try another!

3. Translation by QEMU: Revng

The authors of the open-source tool "revng" have taken a different approach to lifting code. They utilize the CPU emulator often used to host virtual machines (VMs), "QEMU" (Bellard 2005, QEMU 2023). This package, QEMU, is unique in that it allows one to host a VM architecture that is different than the host machine. For example, one can run a Linux virtual machine for the ARM CPU on an intel x86 computer. Another example of cross-CPU emulation is Apple's Rosetta 2 system, allowing a user with x86-64 programs to emulate them on newer Apple computers with their M1 or M2 chips (Evans 2020). Assuming that the processor on the VM is different than on the host, as QEMU encounters instructions they are translated using the Tiny Code Generator (TCG). These individual TCG instructions are then interpreted on the host QEMU system. In this way the "machine" running the VM and the actual machine, running QEMU, need not be the same type of computer; TCG acts as "the middleman" IR. So, "revng" uses the QEMU infrastructure to create TCG, and then to translate the TCG into LLVM. The process starts by handing an executable image to "revng", which uses QEMU to "run" the program. As the internal instructions, in TCG, are encountered, LLVM instructions which match the TCG are emitted.

In the case of this tool the documentation is somewhat better, calling out the prerequisite Linux packages needed, as well as informing the user that they need to install an ancillary tool, "orchestra". This is a command line shell used to set up the proper paths to run the tool. And where to start? With the documented example, which includes the following classical but trivial "Hello World" C test program:

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
   printf("Hello, world!\n");
}
```

Figure 1: Documented test case, from "revng" documentation

After installing "orchestra" and using it to install "revng", as well as a compiler for the ARM architecture, the documentation example compiles the simple program in Figure 1 into an ARM executable program. At this point the executable is translated using the "revng" tool into the native machine (in this case Linux on x86). Next the documentation shows that one can change the permissions on this translated program so that it is executable and run the resulting program natively. With a current Linux version (22.04.1) and following the documentation steps, the resulting translated program simply "core dumps" and exits.

Next the authors noticed an oddity of the translation process. First the "Hello World" program in Figure 1 was compiled twice, once with the GCC compiler and once with the "clang" compiler, into x86-64 binaries. In both cases the programs were compiled with the same options (including "-static") and were not "stripped". The two executable programs are 899 thousand bytes (Kb) and 909Kb, respectively. Compiling the program in Figure 1 into an ARM executable, using the instructions on the repository and the ARM GCC compiler called out in the instructions, generates a file with 123Kb. Lastly, compiling the code on a Raspberry Pi board, which is also ARM and with the same options, gives an executable program that is 517Kb. Note that the x86-64 programs are similar

in size, but the ARM executables are not. The authors did not investigate the reason for this since the second oddity makes it moot.

For the second interesting effect, the authors compiled the program in Figure 1, using "clang", and told the compiler to generate the intermediate LLVM file and stop. The size of this resulting file is 1,781 bytes (not Kb). Following the repository instructions, the ARM executable program was lifted to LLVM. This resulting LLVM file is 4,484Kb, roughly 2,500 times larger.

To attempt to account for this explosion in the size of the resulting translated files, the authors used the Linux tool "objdump" to examine the sections, headers, and disassembly of the contents of the translated executable program. It appears that the result from the "revng" translation is a program designed to interpret the translated IR, not a program translated into actual x86 machine code. For example, there are numerous symbols in the program which begin with "qemu_" such as "qemu_mutex_lock" and others. In addition, functions with names such as "process_pending_signals" implies that the resulting program is designed to run like the original but use the "qemu" infrastructure. Additional clues towards this include many functions with names that start with "target_to_host_" or "host_to_target_". Thus, it appears that the aim is to recreate the same semantics as the original program, but not an actual translation of the original program. This may be incorrect, but it is difficult to tell since the translated program does not run. Time to try another!

4. Success with a Caveat: Retdec

The third binary lifter the authors experimented with is "retdec", the retargetable de-compiler (Avast 2023). The installation instructions for this tool are both complex and simple at the same time; they contain detailed instructions for how to build the programs from scratch on Linux, Windows, MacOS, and FreeBSD. A much easier route is to simply download the compressed Linux "tar" file and extract everything.

The main interface for this tool is a command-line executable program named "retdec-decompiler". The tool has a multitude of command line options, many of which deal with methods for recreating a high-level program code from the original binary. Here the research is to lift a binary to LLVM and stop, but LLVM is an intermediate step in the translation process for this tool. It lifts the binary to LLVM and then converts the LLVM into C code.

An initial test with "retdec" is to take the "Hello World!" program from Figure 1 and run a test that is like the test in the previous section. The extremely simple program is compiled on x86-64 Linux into a Linux executable. The executable is then de-compiled back to C, and a portion of the resulting program is shown in Figure 2:

```
// Address range: 0x1140 - 0x1165
int main(int argc, char ** argv) {
    // 0x1140
    printf("Hello World!\n");
    return 0;
}
```

Figure 2: An initial test of "retdec" decompiler

For trivial code the tool apparently works. However, when decompiling the same program compiled for ARM the results are not quite as great; these results are in Figure 3 and show that the knowledge that the called function is "printf" has been lost somewhere along the line.

```
// Address range: 0x10310 - 0x1031c
int32_t _24_a(void) {
    // 0x10310
    int32_t result; // 0x10310
    return result;
}

// Address range: 0x10430 - 0x10464
int main(int argc, char ** argv) {
    // 0x10430
    _24_a();
    return 0;
}
```

Figure 3: Same program, compiled for ARM and decompiled with "retdec"

When used to recreate a reasonable representation of the original program, at least for x86, the "retdec" tool seems to do a good job and rivals others such as IdaPro and Ghidra in generating appropriate C code (HexRays 2023, Ghidra 2023). For this research the aim is not to decompile back to a high-level language but to lift the binary back to LLVM for potential vulnerability analysis. Thus, the testing of "retdec" in this context is actually concerned, not with the quality of the high-level source code produced, but rather with the quality of the IR created along the way. For this reason no investigation was made into why the ARM code produced was of such low quality.

Similar tests with "retdec" were performed; the simple program was compiled from source code into an LLVM file, and then compiled into an executable which was then lifted back to LLVM. The original compiled file contains 1,423 bytes and the lifted LLVM file contains 4,890 bytes. There is no explosion in size as there was with "revng". Further, a quick look at the reconstructed C source from "retdec" shows why the size is larger. This tool not only attempts to recreate the original program, but also all the ancillary code that is associated with it; this includes the C start-up function "_start" that handles command line parameters, the "_init" and "_fini" functions present in a C/C++ program compiled on Linux, and so on. Figure 4 shows "_init":

```
// Address range: 0x1000 – 0x101b
int64_t _init(void) {
    int64_t result = 0; // 0x1012
    if (*(int64_t *0x3fe8 != 0 ) {
        // 0x1014
        __gmon_start__();
        result = &g3;
    // 0x1016
    return result;
}
```

Figure 4: "retdec" decompiles all executable code, even start-up code

An issue with this "quality" is that one cannot simply take the resulting C program and immediately compile it, as all the extraneous functions will be multiply defined. It's almost "too good" in that sense. Because the ancillary code produced should be the same for each program, the roughly three-times size increase is presumably washed out with a larger executable. To test this hypothesis, the authors selected a slightly more complex program specimen; a program with approximately 200 lines of C code that serves to simulate the functions of a router. When compiled with "clang" to generate LLVM code, the result is 28,550 bytes with 569 lines of IR code. When lifted from the executable the LLVM is 34,226 bytes with 764 lines. A quick examination shows that again this difference is principally the addition of the start-up functions being decompiled as well.

The conclusion of the authors is that with one minor issue, this tool successfully lifts at least x86-64 binaries to LLVM and as a nice side effect generates a quite good representation of the original source code. The one minor issue? The generated IR is for a prior LLVM version and cannot directly be used with a current LLVM package.

5. Microsoft Jumps In: llvm-mctoll

Lastly the authors installed and tested a tool currently available on GitHub called "llvm-mctoll". This tool was written and reported on primarily by Yadavalli (2019) from Microsoft. While the project comes in source code form, the build procedure (unlike many other open-source projects) is largely described in simple terms with minimal mistakes. In fact, the project appears to be very active. As updates to the project are made, the current version of the LLVM software necessary is not only specified in a ".txt" file in the repository but is also a relatively current and supported version of the IR. Additionally it handles the "Hello world" program of Figure 1 with only one caveat, generating a reasonably sized LLVM file as a result.

The caveat mentioned has to do with the call made in Figure 1 to the common C function "printf". Modern CPUs such as x86-64 or ARM call functions and pass the parameters to those functions in CPU registers. It is not always possible to determine what parameters are used at the point of a function call. For instance, the simple "printf" call itself – while it is not shown in the trivial example above – can take a variable number of parameters. Figure 1 only has one, but the function can have an arbitrary number of additional data after the string. It is necessary for a lifting tool to determine which CPU registers are function parameters, as the LLVM "call" instruction requires that these parameters are explicit in the IR. Details of this issue are spelled out by Fink (Fink 2022) but can be summarized; In the process of examining the native instructions, the tool tracks which registers have been assigned to but not changed before the point of the function call. It is significantly more complex than

described here but details are found in the paper by Fink. The caveat referred to above is that this tool requires a user to provide a viable function prototype for any and all functions not known by the tool!

An example of this gotcha is “printf” itself. When lifting a binary, say an x86-64 binary on Linux, it is necessary to tell the tool the format of any call to “printf”. This format is called the function “prototype” and in the case of “printf” is found in a standard file in the compiler infrastructure. In cases where a function is called from the binary code, but the prototype for that function is not known, “llvm-mctoll” stops and will not produce IR at all. All functions must be known by the tool, even if the tool’s user must guess the prototype. Incidentally, the popular reverse engineering tool “ghidra”, developed by NSA, takes a global view across all code in an executable and it propagates the calling convention recovered in any function call across all references to the function (Toor 2022).

The performance of this tool appears to be good and is reported upon in Toor as well. Toor used the Phoenix-2 system, which is an in-memory implementation of the MapReduce data model popularized by Google (Kozyrakis 2023). Unfortunately, this is not a standard benchmark such as SPEC (SPEC 2023) but is fine for reasonable estimates of performance. Details are in the paper but there is reasonably small impact on the size or speed of the lifted (and recompiled) program, assuming that after it is lifted it is optimized by the LLVM tools before being converted back to machine code.

For the practical application desired here, performing vulnerability analysis on LLVM, the question is whether this tool is acceptable. As a test the authors started from one of the SARD vulnerability cases, number 240,080. This file is an example of CWE 415, inadvertently freeing allocated memory twice. The compiled source for this test generates an LLVM file of 272 lines and 10,313 bytes, while the lifted file contains 1,064 lines and 42,922 bytes. This is not an unreasonable expansion. One of the functions within this test case has an example of the vulnerability, and Figure 5 displays the control flow of the bad function from this test case, for the compiled and lifted versions. We conclude that this tool, with supplied function prototypes, will work for our vulnerability scanner.

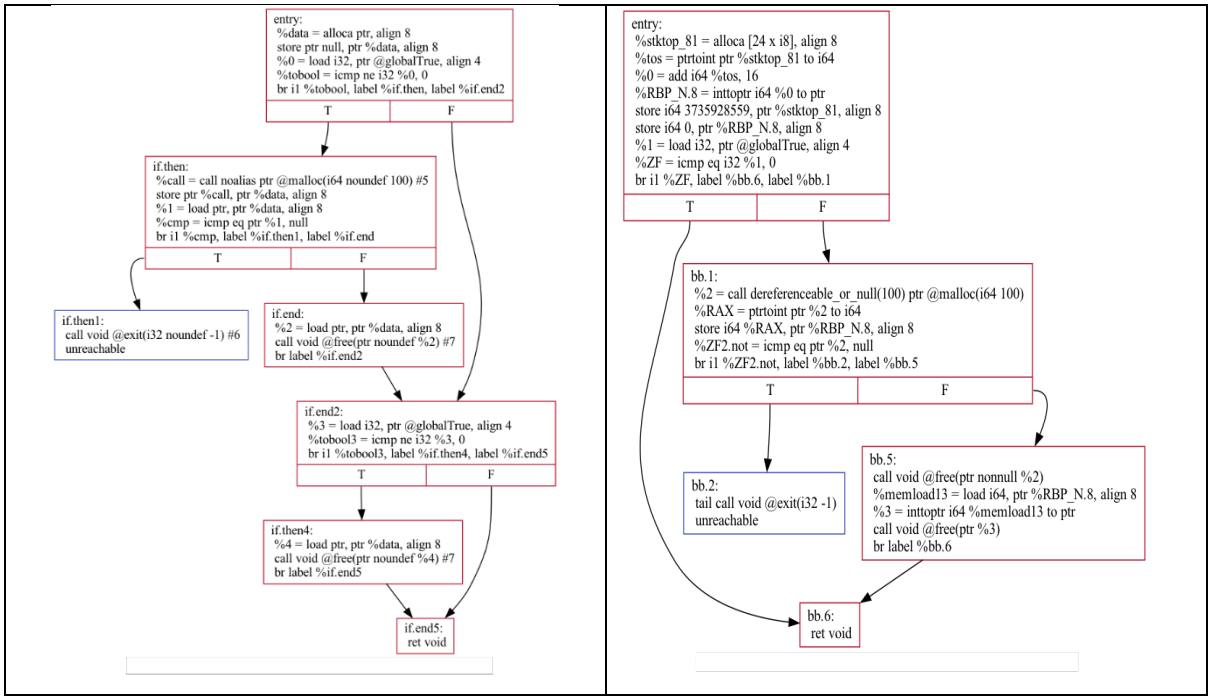

Figure 5: SARD test case “Bad” function compiled (left) and lifted (right)

6. Conclusions and Future Work

Recall that our primary motivation for finding a viable “lifter” for executable programs is the capability to hand the resulting LLVM to the static analysis tools the authors are constructing in a parallel effort. Also recall that the method for testing *that* effort is to compile test cases from the SARD database (SARD 2023). The SARD test cases are intentionally minimal in the sense that they are designed to not require noteworthy external support. Rather, the test cases have a small set of external functions to provide for printing the results of the tests. These few functions, as are the test cases themselves, open for examination. The majority of the test programs needed

for the research effort rely on just a few common files, which contain the necessary prototypes. For example the SARD files “std_testcase.h” and “std_testcase_io.h” have appropriate function headers and thus can simply be includes in the command line for “llvm-mctoll”.

The result is that it is a simple matter to provide function prototypes for the printing capabilities of the test cases. The direct result is that the fourth attempt at locating a “lifter” for our static analysis tool is a successful search; the authors will be proceeding with “llvm-mctoll”.

References

Avast (2023) “RetDec, a retargetable machine-code decompiler based on LLVM”, [online], https://github.com/avast/retdec accessed October 2023.

Crack (2023), [online], available at https://github.com/crack-lang/crack accessed September 2023.

Bellard, F. (2005) “QEMU, a Fast and Portable Dynamic Translator”, Proceedings of the Annual Conference on USENIX (ATEC ’05) , USENIX Association, Berkeley, CA, 41–41.

Evans, J. (2020) “Everything you need to know about Rosetta 2 on Apple Silicon Macs”, [online], Computerworld, November 19, 2020, at https://www.computerworld.com/article/3597949/everything-you-need-to-know-about-rosetta-2-on-apple-silicon-macs.html accessed October 2023.

Fink, Martin (2022) “Translating x86 Binaries to LLVM Intermediate Representation”, Bachelor’s Thesis in Informatics, Technische Universität München, [online], at https://dse.in.tum.de/wp-content/uploads/2022/01/translating_x86_binaries_into_llvm_intermediate_representation.pdf viewed October 2023.

GCC (2023), [online], at https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html accessed October 2023.

Ghidra (2023), [online], at https://ghidra-sre.org and viewed October 2023.

HexRays IdaPro (2023), [online], http://www.hex-rays.com/products/ida/index.shtml accessed September 2023.

Intel (2023), [online], “Intel XED” at https://intelxed.github.io viewed September 2023.

Johnson, S. C (1978) “Lint, a C Program Checker”, Computer Science Technical Report, October 25 1978, Bell Labs, 78-89.

Johnson, S. C. (1979) “Lint, a Program Checker”, Unix Programmer's Manual, Seventh Edition, Vol. 2B, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

Kozyrakis, C. (2023) “Phoenix”, [online], at https://github.com/kozyraki/phoenix accessed October 2023.

LLVM (2023), [online], at https://llvm.org viewed September 2023.

McSema1 (2023), [online], at https://github.com/lifting-bits/mcsema viewed September 2023.

McSema2 (2023), [online], video available at https://www.youtube.com/watch?v=x08mZevfiK8 accessed September 2023.

Mahoney, W., Hoff, G., McDonald, J. T., Grispos, G., (2021) “Software Fingerprinting in LLVM”, 16th International Conference on Cyber Warfare and Security, 2021.

Mahoney, W., Sigillito1, P., Smolinski, J., McDonald, J. T., Grispos (2022) “Analyzing the Performance of Block-Splitting in LLVM Fingerprinting”, 17th International Conference on Cyber Warfare and Security, March 17-18, 2022, Albany, New York, USA.

Mahoney, W., McDonald, J. T., Grispos, G., Mandal, S. (2023) “Improvements on Hiding x86-64 Instructions by Interleaving”, 18th International Conference on Cyber Warfare and Security, March 9-10, 2023, Towson University, Towson, Maryland.

Mitre (2023), “CVE Overview”, [online], at https://www.cve.org/About/Overview viewed September 2023.

PC-lint (2017), [online], at https://pclintplus.com viewed September 2023.

Pony (2023), [online], at https://www.ponylang.io viewed September 2023.

QEMU (2023), [online], at https://www.qemu.org downloaded August 2023.

Recon 2014 (2014), [online], Conference agenda at https://recon.cx/2014/conference.html viewed September 2023

Revng (2023), [online], at https://github.com/revng/revng downloaded August, 2023.

SARD (2023) “NIST Software Assurance Reference Dataset”, [online], at https://samate.nist.gov/SARD/ accessed October 20233.

Spanier, A., Mahoney, W. (2023) “Static Analysis Using Intermediate Representations: A Literature Review”, Proceedings of the 22nd European Conference on Cyber Warfare and Security, Athens Greece, June 22 and 23, 2023, (Digital ISBN 978-1-914587-70-2)

SPEC (2023) “Standard Performance Evaluation Corporation”, [online], at https://www.spec.org/benchmarks.html and accessed October 2023.

Thomson, P. (2021) “Static Analysis: An Introduction. The Fundamental Challenge of Software Engineering is One of Complexity”, ACM Queue, Volume 19, Number 4, 29–41.

Toor, T. S. (2022) “Decompilation of Binaries into LLVM IR for Automated Analysis”, Master of Applied Science in Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada, [online], at https://uwspace.uwaterloo.ca/bitstream/handle/10012/17976/Toor_Tejvinder.pdf?sequence=3&isAllowed=y accessed October 2023.

Yadavalli. S. B., Smith, A. (2019) “Raising binaries to LLVM IR with MCTOLL (WIP paper)”, LCTES 2019: Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, June 2019, 213–218.