Software Compilation Using FPGA Hardware: Register Allocation

Yiming Tan[†], Aditya Diwakar[*‡], Jason Jagielo[†] and Vincent Mooney[*†]
[†]School of Electrical and Computer Engineering, [*]School of Computer Science, [‡]School of Mathematics
Georgia Institute of Technology, Atlanta, Georgia, USA
{ytan308, adiwakar8, jjagielo3, mooney}@gatech.edu

***Abstract*—Malicious attackers are constantly attacking software compilers and attempting to exploit various security vulnerabilities. By executing carefully chosen machine instructions already present in the program, an attacker can perform harmful actions arbitrarily. In this paper, we propose hardware/software codesign techniques to perform software compilation steps in hardware, specifically the register allocation step on a Field Programmable Gate Array (FPGA). Our experiment incorporates two key features: 1) Advanced RISC Machine (ARM) instruction set architecture (ISA)-based register allocation algorithms to calculate variable liveness as well as map virtual registers to physical registers and 2) the feasibility of executing the register allocation algorithms on a Cyclone V FPGA. Our experimental results show the timing efficiency and resource efficiency – while diminishing security risks – when performing register allocation of the `gcd` program on the FPGA.**

***Keywords*—Hardware/Software Codesign, FPGA, Compiler, Register Allocation, Hardware Security**

I. Introduction

As software becomes increasingly complex and ubiquitous, security threats are becoming more prevalent and sophisticated. The majority of cyberattacks today still occur as a result of exploiting software vulnerabilities. Software-based exploitation occurs when certain features of a software stack and interface are exploited [1]. Compiling software is a crucial step in the software development process but also poses security risks, especially for so called "just-in-time" compilers. In a method such as return-oriented programming (ROP), attackers may alter a program's control flow by gaining control of the call stack so that they can cause the program to return to arbitrary points in the program's code. In a ROP attack, the attackers find gadgets within the original program text and cause them to be executed in sequence to perform a task other than what was intended [2]. Attackers can also perform harmful actions arbitrarily by executing carefully chosen machine instructions already present in the program. Moreover, software compilers are also at risk to Trojan source attacks in which adversaries can introduce targeted vulnerabilities into software potentially without being detected by using control characters to modify the order of character blocks displayed and enabling comments and strings to appear to be code and vice versa thereby crafting code that is interpreted differently by compilers as compared to by humans [3][4].

To address these security risks, there has been growing interest in performing compiler steps in hardware, which results in increased security benefits. These benefits can range from increasing the complexity of the attack to limiting overall vulnerabilities.

Our work makes the following contributions to compiling software in hardware:

1) Implementation of register allocation algorithms based on ARM ISA that can be executed on an FPGA given input programs.
2) Hardware/software codesign techniques that use few memory resources and have comparatively high speed.

The rest of the paper is organized as follows. Section II provides insight into prior work and motivation to design a compiler in hardware. Section III provides background on compilation frameworks, back end compilation, register allocation (the main focus of this paper), and rsYocto, an embedded Linux distribution customized for Cyclone V FPGAs. Section IV discusses the methodology of our customized Binary-IR "packet" design, data transmission, algorithms for liveness analysis, register allocation, and our testing example as well as post-allocation processes. Section V describes our Intel DE-10 Standard Board based experimental platform, experimental flow, and design specifics. Section VI discusses the experimental results from the perspective of timing efficiency and resource efficiency. Section VII presents broader discussions of our experimental results while offering future work. Section VIII presents the conclusions of the paper.

II. Prior Work

Previous work using FPGA hardware to compile software includes the demonstration of mitigating shared library function attacks by implementing a hardware root of trust (RoT) from which to store and retrieve function pointers [5]. This prior work from MECO 2019 [5] prevents relocation section overwrites from diverting control flow as they would in an unprotected binary. This was done by implementing a hardware RoT to protect symbol tables from malicious modification and contributing to software libraries to support the storage/retrieval of symbol table entries to/from the proposed hardware RoT, providing embedded system developers with a security measure similar to RelRO without requiring a customized memory management unit [6]. Finally, our algorithm for register allocation is based on a prior syntax-directed translation technique using global optimization, local optimization, code generation, and peephole optimization [7]. We, however, implement this algorithm in FPGA hardware.

Our paper presents the first work to develop the experiment-based register allocation algorithm and implement the register allocator in resource-constrained hardware, specifically on the Cyclone V FPGA. As a crucial step of back end compilation, register allocation maps infinite virtual registers to finite physical registers that are available in a given ISA, seeking to reuse physical registers and minimize the number of memory loads and stores needed by a software program. In this paper, we perform register allocation for a software program using the Cyclone V FPGA to explore the security benefits of hardware-based compilation of software programs.

III. Background

In this section, we discuss the modular compiler design and a critical component of back end compilation: register allocation. Further, we highlight the memory layout constraints for FPGAs and concerns to be addressed when writing memory-dependent applications for an FPGA.

A. Compilation

Compilation is the process of converting source code written in some high level languages such as C or C++ and emitting target specific instructions. Compilers such as Clang for C and C++ utilize LLVM intermediate representation (IR) as an intermediate representation. This exists as a common language for many high level languages to perform optimizations. The LLVM project consists of target specifications that allow the conversion from LLVM IR instructions to target instructions such as ARM or x86 [8]. This style of compiler consists of three parts.

1) Front End: Emits IR from source code such as Clang when using `clang -S -emit-llvm`.
2) Middle End: Optimizations performed on an IR such as dead code elimination, constant propagation, etc.
3) Back End: Given some target specifications (i.e. ARM), emits target assembly given the program in an IR.

This paper focuses on back end compilation and leaves the middle and front end compilation as future work.

B. Back End Compilation

Back end compilation consists of instruction selection, instruction scheduling, and register allocation. While instruction scheduling is not required for correctness, instruction selection and register allocation are required.

- Instruction Selection is the conversion of IR instructions into target instructions such as converting `icmp sgt` and `br` in LLVM IR to `b.gt` in ARM.
- Instruction Scheduling is the process of finding a reordering of instructions to optimize for underlying architectures.
- Register Allocation is the focus of this paper and is the process of converting an IR's use of unlimited virtual registers to a fixed number of physical registers while reducing the need for memory lookups.

C. Register Allocation

There exist various algorithms for register allocation [9]. Many are implemented by the LLVM project in software [8]. This paper covers a two-part algorithm.

1) Liveness (the range during which a variable is being written to or read from) is computed.
2) Register Allocation assigns a physical register to a virtual register for the duration of its live range disallowing that register from being allocated again in this instruction range.

A naive register allocator can be constructed in a way that simply forces a memory lookup for each variable used. This is obviously wasteful, so we implement a register allocator that tends to utilize as many physical registers as possible.

D. rsYocto

rsYocto is an open source embedded Linux distribution designed based on the Yocto project, a classic Linux Foundation collaborative open source project which creates Linux distributions specifically for embedded software that is independent of its underlying architecture. Compared to Yocto, rsYocto has a custom build flow for Intel System-on-Chip (SoC)-FPGAs specifically to customize for the strong requirements of modern embedded SoC-FPGA applications [10].

rsYocto implements interfaces between the hard processor system (HPS) and the FPGA including the advanced eXtensible interface (AXI) to help users interact with HPS hard-IP, FPGA soft-IP, and their peripherals.

IV. Methodology

Our methodology utilizes an Intel DE-10 Standard Board equipped with a dual core ARM-based microprocessor and an FPGA which are connected over a high speed interconnect (AXI) bus all on one chip [11].

Our approach to FPGA register allocation takes as input LLVM IR output from Clang when passed in a test program in source code. This LLVM IR is converted into a fixed-length binary format, similar to a packet used for internet communications, for compilation on the FPGA.

The LLVM IR packet is transmitted to the FPGA using the aforementioned AXI bus to transmit the packet between the microprocessor and FPGA memory. The FPGA analyzes this binary encoding to produce liveness information and finally emit register allocations.

These emitted register allocations are subsequently replaced in the binary format and sent back to the ARM microprocessor over the AXI bus for execution. For larger programs consisting of multiple compilations, linking could be done on the microprocessor. This compilation flow from source to binary is detailed visually in Figure 1.

A. Binary Format

To avoid sending string-based programs to the FPGA for analysis, we design a "Binary-IR" format. This format represents the set of instructions with full granularity needed to compute liveness information and register allocation. Each

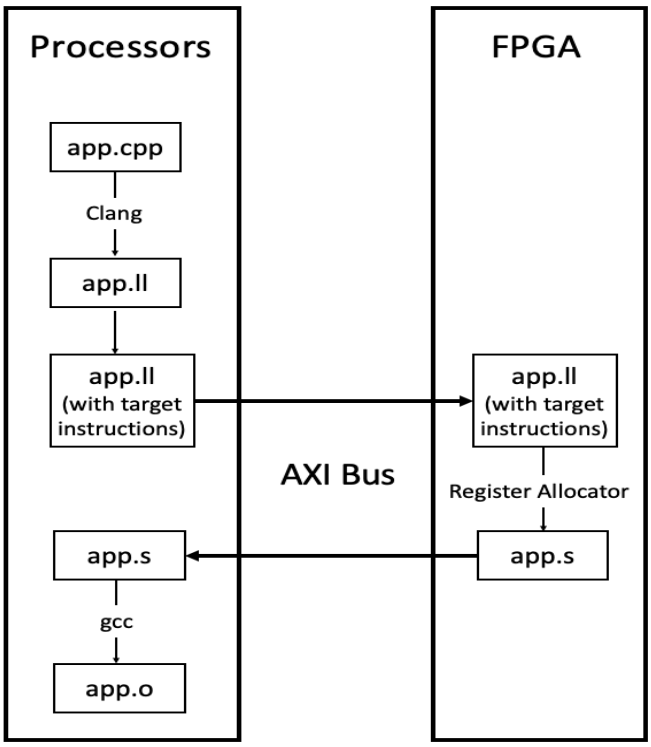

Fig. 1. Visual representation of the methodology from source code to object file with intermediate software compilation steps completed through an FPGA.

instruction consists of 48 bytes of information such as opcode, targets/destinations (if a control flow instruction), and register/-constant operands.

B. Transmission

By utilizing the AXI bus between the processor and the FPGA (Figure 1), the binary-encoded instructions are transmitted to the FPGA for analysis, utilizing a set of tools based on the rsYocto Project.

Specifically, instructions are written into and read from the FPGA memory over the AXI bus using Linux commands executing on the ARM microprocessor. Due to limitations during testing, each instruction of 48 bytes was transmitted to the FPGA memory. This was completed for each instruction by utilizing a Bash script that executes write and read bridge commands to write and validate instructions in FPGA memory.

C. FPGA Analysis

When instructions are received by the FPGA, a two part finite state machine (FSM) begins liveness analysis and register allocation. The implementation of this was first completed in C++ for software parity and subsequently converted to System Verilog to be synthesized to the FPGA.

1) Register Allocation: The register allocation algorithm is as follows:

Algorithm 1: Allocate Virtual Registers

```
Input  : Instructions I, Registers R, Liveness ends
Output: Virtual register allocations
foreach instruction i in I do
    foreach operand op in i not null do
        if op is not allocated then
            for r in R that is available do
                if r is not allocated and r then
                    Allocate op to r
                    Make r unavailable until ends[op]
                    break
```

The pseudocode in Algorithm 1 completes register allocation by iterating through each instruction and finding a register to allocate if the operand has not yet been allocated. This algorithm returns a set of allocations mapping virtual registers to physical registers.

Note that Algorithm 1 refers to information in an $ends$ mapping which is an array of liveness values. This is the information computed during the prior step of liveness analysis.

The pseudocode as described utilizes a set of abstractions that exist in the software world, but typically are not immediately available in the hardware world. Hence, both the software and hardware implementations of this pseudocode are written as a state machine performing a single read or write per state. For loops are unrolled into states with states corresponding to loop condition checking, loop internals, and loop breaking.

2) Liveness Algorithm: Similar to Register Allocation as discussed in the previous section (Section IV-C1), the liveness algorithm was implemented in hardware as well. The algorithm for determining liveness consists of doing a backwards pass on instructions and keeping track of the last used or written location of an operand given any basic block.

The liveness algorithm is implemented as follows:

Algorithm 2: Compute Liveness Information

```
Input  : Instructions I
Output: Liveness information for instructions in I
foreach instruction i in I iterating backwards do
    foreach operand op in i not null do
        if op first instance then
            Mark op last location as i.index
```

In a similar fashion to register allocation, this algorithm is also written in System Verilog as an FSM with loops unrolled into relevant states.

D. Linking and Miscellaneous

Once the register allocations have been made on-board, the completed allocations are available in the FPGA memory for access using the ARM microprocessor interfacing through the AXI bus. This information is then read and combined with the

original program to perform register replacement, and `gcc` is used to convert from assembly to binary.

Additionally, since an object file is created by `gcc`, additional linking can be done. Hence, single functions can be back end compiled on the FPGA and combined with a larger program in software through linking.

An integral test case for this paper is the `gcd` algorithm that computes the maximal number d for two numbers a and b such that $d \mid a$ and $d \mid b$ (i.e., d evenly divides a and d evenly divides b where to divide evenly means to divide with remainder zero).

Listing 1. gcd.c

```
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    } else {
        return gcd(b, a % b);
    }
}
```

After this program is compiled, `gcd.o` can be additionally linked to other programs, for example, a main program that calls `gcd` with parameters a, b and prints the result. This can be done by forwarding the signature of `gcd` to the consuming program using a header file, as the definition would be provided during the linking process (from the `gcd.o` yielded from the aforementioned process). Hence, code bases can be compiled in a hybrid fashion with critical functions compiled in hardware and linked together.

V. Experimental Platform

A. The DE-10 Standard Development Kit

The DE-10 standard development kit is a hardware design platform built around the Intel SoC FPGA. Intel's SoC integrates an ARM-based HPS consisting of a processor, peripherals and memory interfaces tied seamlessly on chip with the FPGA fabric using a high-bandwidth interconnect backbone [12].

1) Intel DE-10 Standard Board: The board used for prototyping the back end of the compiler is an Intel DE-10 Standard Board.

2) ARM-based HPS: The main component of the HPS on the board is a 925MHz dual-core ARM Cortex-A9 MPCore processor.

3) Cyclone V FPGA: The FPGA used on the board is the Cyclone V SoC 5CSXFC6D6F31C6N device, which has 110K programmable logic elements, 166,036 registers, and 5761 Kbits embedded memory (i.e., BRAM) [13].

4) Quartus: To use the board, we use Quartus to compile and synthesize the hardware description language (HDL) code. We program the FPGA using the output SRAM object file (.sof) or binary file (.rbf) generated by Quartus.

B. Experimental Flow

The flow of our experiment is as follows:

1) a laptop with Clang and Binary-IR parser to generate `gcd` Binary-IR from `gcd.c`.
2) HPS with a serial connection to a laptop and the underlying connection to an SD card with rsYocto boot code on the card, enabling transferring of `gcd` Binary-IR from the laptop to the HPS and Linux command execution to interact with the HPS, FPGA, or both.
3) AXI interface to the HPS and to the FPGA, enabling the writes of `gcd` Binary-IR to the memory units on the FPGA and reads of `gcd` ARM assembly back shown in the serial console.

C. Design Specifics

To transfer `gcd` Binary-IR to the memory unit on the FPGA, we first create an AXI interface using Qsys, the platform designer in Quartus. The AXI interface has both FPGA-to-HPS and HPS-to-FPGA widths set to 64 bits and a lightweight HPS-to-FPGA interface with a width of 32 bits to reduce traffic and data transfer time. We also instantiate memory slave components on the FPGA to connect the register allocation logic components with the AXI bus in the form of Parallel Input/Output (PIO) ports or Avalon Memory-Mapped (Avalon-MM) Slave interface, a custom interface for HPS and FPGA communication. This additionally enables the communication to the Block Random Access Memory (BRAM) on the FPGA. After the design flow is successfully set up in Qsys, a binary file (.rbf) is generated by Quartus. We then load this binary file onto the FPGA to enable the processor to read from and write to the FPGA memory units. To automate the interactions between the FPGA memory units and processors, we also have a script to continuously fill in the `gcd` Binary-IR to FPGA memory units. In this way, we send the `gcd` Binary-IR successfully from the processor through the AXI bus to the FPGA and store the Binary-IR code in the FPGA memory unit.

We then implement register liveness and register allocation algorithms in System Verilog, taking in the `gcd` instructions from FPGA memory unit and emitting register allocations. In order to work smoothly with `gcd` ARM assembly, we implement three enumerated data types, respectively, as follows: one for instruction opcode, one for register types (physical register, virtual register, or stack space), and one for the various states of the state machines. Meanwhile, we also define an instruction struct including all important fields of each assembly instruction, such as opcode, source registers, destination register, etc. Register liveness takes in `gcd` instructions, extracts the opcode, destination register, two source registers and iterates through each `gcd` instruction to find where each virtual register ends. This iterative process is done in an 11-state FSM with (i) three states to check whether destination and either of the two source registers exist; (ii) three states to extract register number if the destination register, or either source registers are virtual registers; and (iii) additional states to iterate through and write the liveness information for each virtual register to the FPGA memory unit. At last, the register liveness algorithm yields live ranges for each virtual register, stored in another unit of FPGA memory, which is then read and used by the register allocation algorithm.

Register allocation takes in the `gcd` instructions as well as the live information for each virtual register derived from the register liveness algorithm, the previous stage, to produce the mapping between virtual registers and physical registers. Allocation adds on to the FSM from register liveness for handling the allocation specifics with regard to the destination register, source register 1, and source register 2 respectively. Register allocation has four extra crucial states to find free physical registers in order to assign them to virtual registers: i) one state for checking if the current virtual register has already been mapped to a physical register; ii) one state for checking if any physical register is available; iii) one state for checking which physical register is available specifically next in line; and iv) one state for recording the mapping between the current virtual register and the available physical register.

Since there are only 13 general purpose physical registers available in the ARM ISA, if the input program at some point uses up all 13 physical registers, through iterations in the four states mentioned above, we can map the current virtual register to a memory location instead of a physical register, so that we can spill the value held by the current virtual register to memory (as stack space). Moreover, register allocation algorithm also has the functionality to free the physical registers that were allocated to virtual registers that are no longer live. Therefore, our algorithm can largely minimize the physical register usage, reduce latency, and increase program efficiency. The output of this FSM is a mapping between all virtual registers used in the `gcd` instructions and their corresponding physical registers, stored in the memory unit on the FPGA.

After we obtain the `gcd` assembly output with physical registers, we convert the `gcd` assembly to binary using `gcc` and perform additional linking as necessary.

Finally, the `gcd.o` is executed on HPS, where the output is sent through the serial interface to the computer serial console, concluding the experiment.

VI. Experimental Results

Our register allocator is able to successfully 1) analyze register liveness information, 2) allocate a minimal number of physical registers for the `gcd` program on the FPGA, and 3) complete various examples of the `gcd` program based on different user inputs. The instructions of the `gcd` program range from stores, loads, addition, subtraction, branches, etc, and our testing examples cover the longest control flow path over the `gcd` program. All the measurements shown in this section are based on testing examples of the `gcd` program which goes through a long and significant control flow path, in other words, using all of the basic blocks in our algorithm. Through thorough testing, the correctness of our algorithm proves the feasibility of FPGA-based software compilers and more importantly under the condition of low hardware resource utilization and comparatively low latency.

Table I summarizes the resource utilization when performing register allocation of a `gcd` example at a 50 MHz clock rate on the FPGA. We observe $< 0.5\%$ of resource utilization for register liveness analysis and $< 2\%$ of resource utilization for register allocation algorithm in terms of both logic and registers. This shows our proposed algorithm and its implementation in System Verilog is extremely resource-efficient, enabling further applications of the register allocator on resource-constrained devices.

TABLE I
Resource Utilization on Cyclone V FPGA

Processes	Resources	Utilization	Utilization%
Register Liveness	Logic (in ALMs)	184	0.44
	Registers	342	0.41
Register Allocation	Logic (in ALMs)	796	1.89
	Registers	1249	1.49

TABLE II
System Verilog Compilation Time of Register Allocation Algorithms

Processes	Total Time
Register Liveness on FPGA	78 s
Register Allocation on FPGA	117 s

The average compilation time of the register allocation in System Verilog is illustrated in Table II. Total hardware synthesis time refers to the time of the following steps in Quartus: *Analysis and Synthesis*, *Fitter*, *Assembler*, *Timing Analyzer*, and *EDA Netlist Writer*. As a necessary step for the FPGA, *Fitter* occupies a significant portion of the Quartus compilation process for System Verilog and takes a significant amount of time.

TABLE III
Average Execution Time of Register Allocation Algorithms

Processes	Time
Register Liveness on FPGA	10.34 μs
Register Allocation on FPGA	8.74 μs

We also measure the average execution time of the register allocation algorithms based on testing an example going through the longest control flow path of the `gcd` program. Results in Table III show the average execution time of register liveness and register allocation algorithms: $10.34\mu s$ and $8.74\mu s$ respectively.

TABLE IV
Average Execution Time on AXI Bus

Process	Time
AXI Write and Read BRAM of 1 Word	23 μs
AXI Write and Read BRAM of 10 Words	114 μs
AXI Write and Read BRAM of 100 Words	857.1 μs

More importantly, as an indispensable part of the execution time measurement, we measure the time for data transmission between the HPS and the FPGA through the AXI bus and directly to or from the BRAM through the Avalon-MM interface in Table IV, ranging from the data size of 1 word, 10 words, to 100 words.

Algorithm 3: Write to / Read from BRAM through AXI Bus

Input : Address A, DataWrite Dw
Output: DataRead Dr
foreach *data* d *in* Dw ***and*** *address* a *in* A **do**
  Write d to a
foreach *address* a *in* A **do**
  Read data from a
  Collect the read data in one set Dr
foreach *data* d *in* Dw **do**
  Perform arithmetic and/or logic operation on d
  Repeat write and read (the first two loops)

In our measurement, we continuously write data to BRAM on the FPGA through the AXI bus and read it from the HPS. We also experiment on performing arithmetic and logic operations with the data transferred before writing to other BRAM locations.

VII. Discussion and Future Work

Through our experiment, we demonstrate that the register allocation steps can be fully performed in hardware on the Intel DE-10 Standard Board.

Although our experiment is based on the Cyclone V FPGA, it can also be emulated on all the other FPGAs, including Xilinx FPGAs. Since both the register allocation algorithms and our hardware/software codesign techniques are FPGA manufacturing agnostic, FPGA choice will not significantly affect the correctness of our experiment, the results of compilation, the timing or resource efficiency of the execution.

However, the execution time and resource utilization may vary as the architecture, functional units, numbers of configurable logic blocks and sizes of BRAMs may vary between different FPGAs. Therefore, with more powerful FPGAs, for example the Xilinx Virtex UltraScale+ FPGA VCU 118 FPGA, which features 2,586K system logic cells and 345.9 mega-bits BRAM, we would expect to see a reduced execution time as well as an advanced hardware environment supporting larger scale and more complicated programs.

Our planned future work includes automating the register allocation process fully on the Intel DE-10 Standard Board. The automation includes the following: 1) use `Clang` to emit `app.ll` from `app.cpp` on the HPS of the board, 2) emit instructions in binary-IR format from `app.ll` on the HPS, 3) transfer instructions in binary-IR format from the HPS to the FPGA interfacing through the AXI bus, 4) register allocation algorithms in System Verilog which read the binary-IR instructions and perform physical register replacement on the FPGA, 5) generate the assembly file `app.s` from instructions with physical registers on the FPGA, 6) send `app.s` from the FPGA to the HPS, 7) use `gcc` to generate `app.o` on the HPS, and 8) link then execute `app.o` on the HPS. The automation of these eight steps will provide a fully functional register allocator only using the Intel DE-10 Standard Board with both software and hardware embedded. Moreover, this work can also be generalized to support the register allocation for all software programs (in addition to the `gcd` program in the experiment).

In addition to register allocation, we also plan to implement a fully functional compiler back end on the Intel DE-10 Standard Board, including instruction selection, instruction scheduling and register allocation to extend and further optimize our current work in terms of resource and timing efficiency.

For more impact, middle and front end compilation can also be performed in hardware, specifically on the FPGA, in order to introduce a fully functional hardware compiler on the FPGA to further secure the compilation process from malicious attackers.

VIII. Conclusions

In conclusion, we have shown a proof-of-concept demonstration of how to compile software on an FPGA which cannot be attacked at run-time (i.e., the FPGA used is not dynamically reconfigurable). Specifically, we have implemented the back end register allocation step in System Verilog and have compiled a `gcd` program to ARM assembly. This research aims to enable full just-in-time compilation on an FPGA at run-time which is protected from cyberattack by implementation in hardware.

References

[1] J. Jang-Jaccard and S. Nepal, "A survey of emerging threats in cybersecurity," *Journal of Computer and System Sciences*, vol. 80, no. 5, pp. 973–993, 2014.

[2] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, Aug. 2014, p. 385–399.

[3] J. Coker, "Most computer code compilers vulnerable to novel attacks," 2021, last accessed 29 Apr 2023. [Online]. Available: https://www.infosecurity-magazine.com/news/computer-code-compilers-attacks/

[4] N. Boucher and R. J. Anderson, "Trojan source: Invisible vulnerabilities," *Computing Research Repository (CoRR)*, vol. abs/2111.00169, 2021. [Online]. Available: https://arxiv.org/abs/2111.00169

[5] G. Lopez, M. Foreman, A. Daftardar, P. Coppock, Z. Tolaymat, and V. J. Mooney, "Hardware root-of-trust based integrity for shared library function pointers in embedded systems," in *8th Mediterranean Conference on Embedded Computing (MECO)*, 2019, pp. 1–6.

[6] P. H. Coppock, M. K. Yacoub, B. L. Qin, A. J. Daftardar, Z. Tolaymat, and V. J. Mooney, "Hardware root-of-trust-based integrity for shared library function pointers in embedded systems," *Microprocessors and Microsystems*, vol. 79, p. 103270, 2020.

[7] K. Keutzer and W. Wolf, "Anatomy of a hardware compiler," *SIGPLAN Not.*, vol. 23, no. 7, p. 95–104, Jun 1988. [Online]. Available: https://doi.org/10.1145/960116.54000

[8] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of International Symposium on Code Generation and Optimization*, Mar 2004, pp. 75–86.

[9] K. D. Cooper and L. Torczon, "Chapter 13 - register allocation," in *Engineering a Compiler (Third Edition)*, K. D. Cooper and L. Torczon, Eds. Philadelphia: Morgan Kaufmann, 2023, pp. 663–712.

[10] R. Sebastian, "rsyocto," 2021, last accessed 29 Apr 2023. [Online]. Available: https://github.com/robseb/rsyocto

[11] Terasic, "De-10 standard user manual," 2017, last accessed 29 Apr 2023. [Online]. Available: https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/Boards/DE10-Standard/DE10_Standard_User_Manual.pdf

[12] J. Fan, "Terasic de10-standard development kit," 2019, last accessed 29 Apr 2023. [Online]. Available: https://www.rocketboards.org/foswiki/Documentation/DE10Standard

[13] Intel, "Cyclone v device overview," 2018, last accessed 29 Apr 2023. [Online]. Available: https://www.intel.com/programmable/technical-pdfs/683694.pdf