University of Central Lancashire UCLan

Central Lancashire Online Knowledge (CLoK)

Title	Towards Anomaly Detection in Embedded Systems Application Using LLVM Passes
Type	Article
URL	https://clok.uclan.ac.uk/52745/
DOI	
Date	2024
Citation	Ilahi, Sirine, Omotosho, Adebayo and Hammer, Christian (2024) Towards Anomaly Detection in Embedded Systems Application Using LLVM Passes. 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC). pp. 2453-2458. ISSN 2836-3787
Creators	Ilahi, Sirine, Omotosho, Adebayo and Hammer, Christian

It is advisable to refer to the publisher's version if you intend to cite from the work.

For information about Research at UCLan please go to http://www.uclan.ac.uk/research/

All outputs in CLoK are protected by Intellectual Property Rights law, including Copyright law. Copyright, IPR and Moral Rights for the works on this site are retained by the individual authors and/or other copyright owners. Terms and conditions for use of this material are defined in the http://clok.uclan.ac.uk/policies/

Towards Anomaly Detection in Embedded Systems Application Using LLVM Passes

Sirine Ilahi
University of Passau
Passau, Germany
sirine.ilahi@uni-passau.de

Adebayo Omotosho
University of Central Lancashire
Preston, United Kingdom
aomotosho@uclan.ac.uk

Christian Hammer
University of Passau
Passau, Germany
christian.hammer@uni-passau.de

***Abstract*—Software security exploits, such as Return-Oriented Programming (ROP) attacks, have persisted for more than a decade. ROP attacks inject malicious behaviors into programs, posing serious risks to computing devices, and they can be particularly challenging to detect in systems with limited resources. In this paper, we introduce an approach that exploits Low-Level Virtual Machine (LLVM) passes, programmatic transformations applied during compilation, to detect ROP attacks in ARM-based embedded systems. By customizing LLVM passes, developers can integrate tailored security checks and optimizations into embedded systems requirements. Our approach is motivated by the use of Hardware Performance Counters (HPCs) for certain mitigations, which are not commonly available on all embedded systems. The experimental evaluation of our approach for detecting ROP attacks in real-world applications shows that it is feasible and can be extended to detect new attacks independently of an Operating System (OS). The storage overhead induced by our approach is approximately 55%.**

***Index Terms*—Embedded systems, LLVM passes, Instrumentation, Return oriented programming, Security.**

I. Introduction

Embedded systems typically lack the robust security mechanisms provided by an Operating System (OS). Traditional security techniques, such as address space layout randomization (ASLR) or data execution prevention (DEP), are frequently absent or difficult to implement [1].

In addition, Hardware Performance Counters (HPCs) based approaches that count certain hardware events to distinguish vulnerable and benign applications have been well investigated [2]–[6]. These statistically based approaches have been reported to be computationally cost-effective, however, HPC registers are not available on all embedded devices [7]–[9].

In this paper, we investigate a software-based counter approach derived from Low Level Virtual Machine (LLVM), a popular compiler infrastructure that offers a range of tools and technologies for code optimization, analysis, and transformation. At the core of this infrastructure is a language-independent Intermediate Representation (IR), similar to a portable, high-level assembly language. This IR can be optimized through multiple passes which makes LLVM highly versatile. One of the key components of LLVM is the LLVM pass, which is responsible for performing the specific transformations on the IR code. In the context of security, LLVM pass transformation can overcome these challenges by examining the code at the IR level, enabling the detection of vulnerabilities that might be missed by other tools or techniques.

The contributions of this paper are:

- Investigation of the applicability and effectiveness of an LLVM pass to identify vulnerabilities in embedded systems applications.
- Development of a novel technique to detect ROP attacks.
- Performance evaluation of the purposed technique on real-world applications.

II. Background

In this section, we briefly introduce the key concepts related to this paper's scope of work and contributions. More particularly, we describe the memory corruption exploitation technique known as Return-Oriented Programming and detail some ARM features which are used later in the paper.

A. Return Oriented Programming on ARM

Return-oriented programming (ROP) was proposed by Shacham in 2007 for the x86 architecture, and then subsequently extended to the ARM, SPARC, and other processors [10]. ROP attacks are increasingly used in practice, in particular, the recent ROP-based attacks on well-established products such as Adobe Reader, Adobe Flashplayer, or Quicktime Player [11]. The main idea of ROP is to exploit memory vulnerabilities in a program without injecting new code into the program's address space. In a ROP attack, short code snippets in the target program to be executed, so-called *gadgets*, will be chained together in an order designed by the attacker to perform a malicious functionality. Typically each gadget ends with an indirect jump (e.g., *ret* *rcx or *jmp* *rcx) will be chosen to form the ROP code. The attacker chains gadgets together by controlling the target of a gadget's indirect jump to point to the beginning of the next gadget in the sequence.

In Fig. 1, we give an example of ROP exploit that presents a buffer overflow caused by an unchecked *strcpy* that allows the user to smash the contents of the stack and start the execution of the ROP chain at return time.

B. Defences Against Return-Oriented Programming

Despite extensive research in computer security, vulnerabilities caused by memory corruption in low-level programming

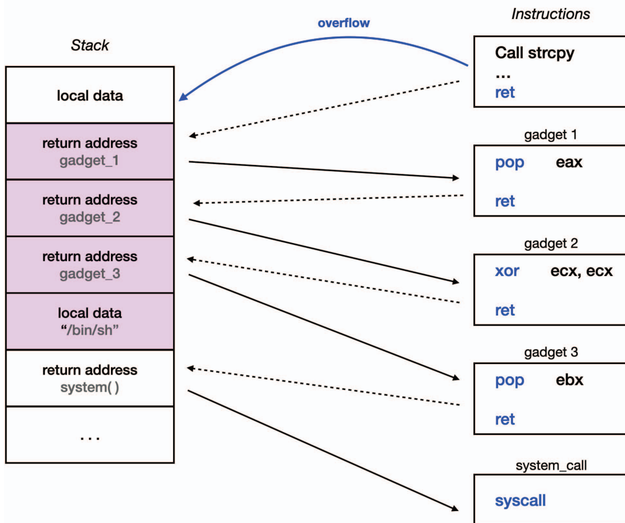

Fig. 1. General principle of ROP attacks.

languages remain to pose a major attack vector. The research community has proposed various approaches to improve memory protection.

1) Data Execution Prevention: Data Execution Prevention (DEP) is is one of the major countermeasures against code injection widely used in modern OS. Its main function is to enforce the restriction that a memory region cannot simultaneously be both writable and executable, and thus prevent the execution of unauthorized code [12]. In the absence of such mitigation techniques, the program could potentially write CPU instructions into a memory segment designated for data and subsequently execute those instructions. While this technique effectively mitigates against code injection attacks, it remains susceptible to return-into-libc attacks that leverage existing code rather than injecting their own [13].

2) Address Space Layout Randomization: Address Space Layout Randomization (ASLR) is another relevant and widely deployed technique aimed at enhancing security against various types of buffer overflows and to render their exploitation more difficult by randomizing the locations of the most important program components in (virtual) memory. Once ASLR is implemented, it is hard to determine the location of the program data. However, despite its effectiveness, ASLR also has its limitations. Research by Schacham [14] highlights a significant weakness in ASLR for both 32-bit and 64-bits architectures: the finite number of bits available for address randomization. This limitation means that only a portion of the address space can be randomized, leaving some bits susceptible to brute-force attacks [15]. Furthermore, another critical ASLR weakness is its susceptibility to memory disclosure attacks, where the adversary gains knowledge of a single runtime address and then uses that information to re-enable code reuse [16].

3) Control Flow Integrity: Control Flow Integrity (CFI) is also another defence technique that restrict the set of possible control-flow transfers to those that are required for correct program execution from being transferred to unintended and malicious addresses [17], making it significantly harder to perform such attacks. By ensuring that program execution follow a valid path through the static Control-Flow Graph (CFG), CFI ensures that the program follows its expected behaviour and prevents deviations [17], which are considered as CFI violation that terminates the application. The goal of CFG is to restrict the set of possible control-flow transfers to those that are strictly required for correct program execution. This prevents code-reuse attacks such as ROP, because they would cause the program to execute control-flow transfers, which are illegal under CFI technique. However, implementing CFI can include an optimization between security and performance. For example, some implementations may sacrifice certain checkpoints by not instrumenting function calls and paying attention only to return instructions.

Nevertheless, this approach can leave vulnerabilities in the protected program, as attackers may exploit remaining widgets. As shown in Coudray *et al.* [18], many CFI implementations have been tested and found to be quite permissive, so an attacker can still carry out ROP attacks despite the security measures in place.

As far as we know, the ROP mitigation techniques we have discussed can be bypassed by knowledgeable adversaries using generic methods.

Fig. 2. Principle design of a three-phase compiler.

III. DESIGN AND IMPLEMENTATION

A compiler is an essential part of software development. It converts source code instructions into object code instructions. Conceptually, its software design consists of three phases: Front-end, Optimizer, and Back-end [19]. Fig. 2 shows the main components of a three-phase compiler design.

In embedded systems where memory resources are often constrained, and energy efficiency is a priority, the ARM processor stands out in a unique way to optimize program size and memory usage [20]. It has become one of the most widely used processors in the world. However, the widespread use of ARM-based devices has led to a significant increase of attacks on these devices. One such attack that has gained popularity is ROP. Based on its requirements, a ROP payload has the two low-level properties:

- a sufficiently long chain of gadgets with few instructions in each gadget.
- a mispredicted return for each gadget's terminal indirect jump instruction.

These properties are intrinsic to each ROP payload and are independent of the program being monitored. To effectively

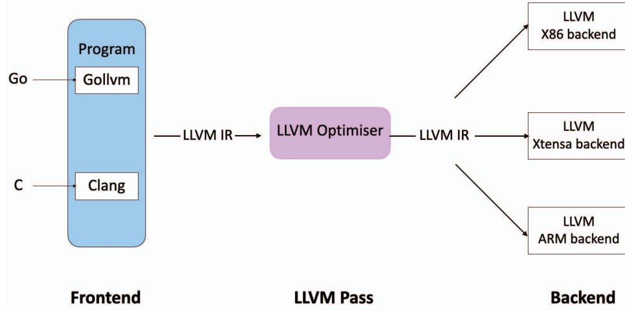

Fig. 3. LLVM inftrastructure overview.

```
if (StoreInst *LI = dyn_cast<StoreInst>(this))
{
  //instrument analysis code
}
```

Listing 1. Address Sanitizer LLVM pass

```
if (StoreInst *LI = dyn_cast<StoreInst>(this))
{
  if (LI->getMetadata("store") && LI->isVolatile())
    {
      //instrument analysis code
    }
}
```

Listing 2. LLVM Instrumentation

target ARM processors under attacks, a comprehensive understanding of assembly language programming is crucial. Indeed, it is not enough to write attacks in a "simple" scripting language; a deeper understanding of ARM binary flow analysis, customized ARM shellcode creation, and ARM program debugging is essential.

A. Design

Code instrumentation is a common technique used to track application behaviour by inserting specific code, called instrumentation code, into the source files under analysis. These files are subsequently compiled and executed. Therefore, the execution output includes this instrumentation code can be be used for further analysis. The most popular usages for code instrumentation are software debugging, monitoring, performance analysis, and aspect oriented programming.

1) IR: An Intermediate Representation (IR) is a data structure that is designed as an internal form that the compiler or the virtual machine uses to represent the source code [21]. Typically, compilers generate the IR as a bridge between the source code and the compiled binaries to generate a generic assembly code. Therefore, a universal assembler can be used to to perform the final conversion from assembly to machine-independent intermediate code. Otherwise, a complete native compiler for different languages and machine architectures would be required.

2) LLVM: The Low Level Virtual Machine is a compiler infrastructure designed to create an interface to the compilation process so that further optimizations can be applied to the binary itself, rather than using Just-In-Time (JIT) compilers for runtime optimizations [22]. Otherwise, the target program can be operated through a chain of analyses and transformations. Each of these steps is called a pass, as illustrated in Fig. 3. A pass performs the transformations and optimizations that form the core of the compiler. The results of these analyses serve as a basis for further transformations [23]. Listing 1 is an example of the Address Sanitizer LLVM pass. This pass only instruments the IR only if it represents a *store* instruction. With a simple check, the modified Address Sanitizer LLVM pass in Listing 2 will only instrument the store instruction if the IR is annotated with *store*, meaning that it is generated from the guest binary.

B. Implementation

In our instrumentation approach, applications are tracked at runtime by by inserting *hooks* into the source code to record events such as the number of entries and exits of each basic block, a sequence of sequential instructions without branches, along with its execution count. These hooks call the monitor for each such event, a process known as instrumentation as shown in Fig. 4. Fig. 4 (a), showcases an unedited version of a simple code example. This example includes a simple recursive function *func()* that calls itself with a decreasing argument *i* until this argument is less than or equal to 0, after which *main()* function calls *func()* with values from 0 to 9. While Fig. 4 (b) illustrates the instrumented source code with added calls to monitor the number of entries and exits for each basic block and the total number of basic blocks executed. Whenever an application calls a function, it calls the ENTER hook and passes control flow to the monitor. The monitor stores the relevant information, such as the number of basic block entries, before returning control to the application for continued execution. Similarly, the monitor manages the EXIT of each basic block at the function's end.

C. ROP exploit creation on ARM

In this section, we describe the leveraged ROP attacks against vulnerable real-world applications on the ARM platform. To test the effectiveness of our approach, we have implemented various ROP attacks which are publicly available on Github taken from Welearegai *et al.* [24]. The total size of the implemented ROP applications is 139.88MB. During the ROP attack, the gadget addresses are loaded into the Program Counter (PC) register using *pop* instructions. The contents of the function argument registers (i.e. r0-r4) must be reserved before the control flow is redirected to the desired function to provide the function arguments. For example, to open a system shell, the r0 register must point to */bin/sh* before the control flow is redirected to the address of the *system* function.

IV. Evaluation and discussion

In this section, we delve into the research questions that arouse our interest, the evaluation methodology employed, the experimental setup, and the results of assessing the LLVM

```
int main(){

    int i;
    for (i=0; i<10; i++)
    {
      func (i);

    }

    return 0;
}
void func (int i)
{

    if ( i>0 )
    {
       func ( i - 1);

    }

}
```

(a)

```
int BASIC_BLOCKS ;
int main(){
    ENTER ("main");
    int i;
    for (i=0; i<10; i++)
    {
      func(i);
      BASIC_BLOCKS++;
    }
    EXIT ("main");
    return 0;
}
void func (int i)
{
    ENTER ("func");
    if ( i>0 )
    {
       func ( i - 1);
      BASIC_BLOCKS++ ;
    }
    EXIT("func");
}
```

(b)

Fig. 4. (a) Original source code. (b) Annotated source code.

pass. The research questions we address are outlined as follows:

RQ1: Can LLVM pass software-based metrics be used for distinguishing ROP attack behavior?

RQ2: How significant is the resource usage of the proposed approach?

A. Evaluation Approach

To answer the first research question, obtaining an accurate record of the software performance counter metrics is crucial. To achieve this goal, we performed experiments involving instrumented code, where supplementary code segments were inserted at the entry and exit points of each basic block. This process was facilitated by the implementation of an LLVM pass as described in *Algorithm 1*. Within this implementation, three global variables *BasicBlockCounter*, *EntryCounter*, and *ExitCounter* were created. These variables respectively track the number of basic blocks, the number of entries, and the number of exits. For each basic block, an instrumented code was added to count its entries and exits, and the counter *BasicBlockCounter* was incremented by 1. Discrepancies in the values of these three counters indicate illegal control flow violations. The pass includes 55 lines of C++ code designed to track these three counters. Discrepancies in entry and exit counts indicate the occurrence of ROP attack. Our instrumentation process was implemented into the LLVM 11.0.1 compiler and the evaluation hardware setup consists of a Raspberry Pi 4 Model B running kernel version 5.4. The approach to the second research question involves measuring the efficiency of the proposed method using metrics such as storage and runtime overhead.

Algorithm 1: Algorithm to instrument LLVM IR and count basic blocks, entries, and exits

```
Input  : LLVM IR
Output: Instrumented IR with number of basic blocks,
        entries, and exits
# initialize variables;
BasicBlockCounter = 0;
EntryCounter = 0;
ExitCounter = 0;
for each Function F in Module M do
    for each BasicBlock B in Function F do
        BasicBlockCounter++;
        InsertEntryCounterCode(B);
        EntryCounter++ ;
        InsertExitCounterCode(B);
        ExitCounter++;
    end
end
for each BasicBlock B in Function F do
    return (BasicBlockCounter, EntryCounter,
     ExitCounter);
end
```

B. Discussion

This section presents our findings based on the two research questions.

RQ1: *Can LLVM pass software-based metrics be used for distinguishing ROP attack behavior?*
From the analysis of our test applications, we identified a key mismatch between ROP attacks and non-ROP implementations, specifically in the differences between the number of entries and exits of basic block counters, as shown in Table I. For instance, in the application *Crashmail*, during normal execution, the number of entry and exit basic blocks is 15, because in normal execution each basic block has its entry and exit. However, when illegal code execution occurs in *Crashmail* due to ROP attack, the number of exits (11) and entries (12) do not match. This discrepancy can be explained by the program's behavior when it jumps to unknown memory locations so that the subsequent exit cannot be counted. The results in Table I show that our approach can correctly distinguish between ROP and non-ROP behaviors in real applications albeit the attacks are only detected after the application terminates.

RQ2: *How significant is the resource usage of the proposed approach?*
Table II presents the overhead of our proposed approach in terms of runtime and storage overhead. The size of the instrumented binary files increases by an average of 55% compared to the original files due to the additional integrated instrumentation code. In terms of runtime, our approach requires an additional initialization overhead of 20ms on average compared to the original code. The increase in execution time ranges from 3.0% to 15.5% depending on the characteristics of the target program and the number of basic blocks on each

TABLE I
COMPARISON BETWEEN ROP AND NON-ROP

Applications	ROP execution			Non-ROP execution		
	#Basic blocks	#Entries	#Exits	#Basic blocks	#Entries	#Exits
Crashmail	12	12	11	15	15	15
Dnstracer	5	5	4	5	5	5
Mcrypt	13	13	12	13	13	13
Nethack	15	15	14	15	15	15
PHP	9	9	8	11	11	11
Wifirx	15	15	14	18	18	18

TABLE II
BINARY SIZE AND RUNTIME OVERHEAD

Applications	Binary size (bytes)			Runtime (ms)		
	Before instr	After instr	Increment(%)	Before instr	After instr	Increment(%)
Crashmail	8464	13128	55.2	257	293	14.0
Dnstracer	6020	7544	25.3	328	338	3.0
Mcrypt	6904	10580	53.3	251	262	4.4
Nethack	8336	13004	56.0	237	259	9.3
PHP	8652	13424	66.7	601	694	15.5
Wifirx	8572	13240	54.6	593	612	3.2

TABLE III
COMPARATIVE ANALYSIS OF OUR APPROACH AND ALTERNATIVES

Characteristics	Our approach	Honeygadget	SafeLLVM	RIO
OS requirement	No	No	No	No
ROP attack detection	Detects ROP attack by differencing the number of entries and exits of basic blocks	Prevents ROP attacks by enforcing return address checks at runtime	Prevents ROP attacks by minimizing the number of gadgets present in binaries using llvm	Prevents ROP attacks by encrypting all return instructions
Average storage overhead	55%	-	-	30%
Average runtime overhead	8.22%	6.8%	0.2%	8.79%

function of the program. The resource consumption of our method is comparable to that of similar existing works, thus validating its practicability in detecting anomalies in embedded systems applications.

V. RELATED WORK

This section discusses some of the closely related approaches to ours and Table III presents the comparison of our technique to some of the existing LLVM works for detecting ROP attacks. Honeygadget [25] consists of inserting *honey gadgets* into the application as decoys to confuse adversaries with traps based on the LLVM pass. However, the enrichment of the type and location of the inserted gadgets is limited as some types of gadgets are rare but necessary for a certain kind of code reuse attacks which consequently limits the types of these honey gadgets and increases the possibility of leaking the traps. In contrast, our approach takes a more general stance as it is not bound to specific gadget types, which makes it adaptable to a wider range of scenarios. Furthermore, Safellvm [26] is a technique that focuses on minimizing the number of gadgets in x86-64 binaries compiled with the LLVM infrastructure. It is able to reduce the number of gadgets in a binary and in most cases prevents the automatic generation of ROP chains. However, this approach has some limitations, as it targets only x86_64 binaries. Moreover, the Return Instruction Obfuscation (RIO) technique [27] encrypts all return instructions and instruments the necessary modules to decrypt and execute the encrypted return instructions using LLVM pass. However, this technique also requires a high storage overhead which is about 30%, to prevent attackers from collecting gadgets. Eberius *et al.* [28] approach is similar to ours in terms of using software-based performance counters, although it targeted exposing low-level open MPI performance information. Their approach basically introduces a simple low-level approach that instruments the Open MPI code at key locations to provide fine-grained MPI performance metrics. Unlike in our paper, their approach did not focus on attack detection but only on how to use software performance metrics to determine bottlenecks in user code and MPI implementation.

VI. CONCLUSION

Memory corruption attacks arising from stack buffer overflow have been a major security problem for decades and have been extensively studied in the academic community. Because these attacks are persistent, this paper investigates innovative software performance counters-based statistical methods using LLVM pass to detect ROP in embedded systems. Although our current approach is limited in that it cannot perform runtime

prevention, our experimental result shows that it can distinguish ROP from non-ROP executions in real-life applications. Hence, our follow-up research will address two objectives: investigating additional LLVM instrumentation counters that can help with runtime prevention of ROP attacks and reducing storage overhead.

REFERENCES

[1] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 31–46.

[2] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 483–493.

[3] X. Wang, C. Konstantinou, M. Maniatakos, R. Karri, S. Lee, P. Robison, P. Stergiou, and S. Kim, "Malicious firmware detection with hardware performance counters," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 3, pp. 160–173, 2016.

[4] S. P. Kadiyala, P. Jadhav, S.-K. Lam, and T. Srikanthan, "Hardware performance counter-based fine-grained malware detection," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 5, pp. 1–17, 2020.

[5] K. Basu, P. Krishnamurthy, F. Khorrami, and R. Karri, "A theoretical study of hardware performance counters-based malware detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 512–525, 2019.

[6] C. Li and J.-L. Gaudiot, "Detecting spectre attacks using hardware performance counters," *IEEE Transactions on Computers*, vol. 71, no. 6, pp. 1320–1331, 2021.

[7] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *Proceedings of the 2018 on Asia conference on computer and communications security*, 2018, pp. 457–468.

[8] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proceedings of the sixth ACM workshop on Scalable trusted computing*, 2011, pp. 71–76.

[9] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 20–38.

[10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 559–572.

[11] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011, pp. 40–51.

[12] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha, "Launching return-oriented programming attacks against randomized relocatable executables," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2011, pp. 37–44.

[13] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, "Marlin: A fine grained randomization approach to defend against rop attacks," in *Network and System Security: 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings 7*. Springer, 2013, pp. 293–306.

[14] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 298–307.

[15] V. Parikh and P. Mateti, "Aslr and rop attack mitigations for arm-based android devices," in *Security in Computing and Communications: 5th International Symposium, SSCC 2017, Manipal, India, September 13–16, 2017, Proceedings 5*. Springer, 2017, pp. 350–363.

[16] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE symposium on security and privacy*. IEEE, 2013, pp. 574–588.

[17] Y. Ruan, S. Kalyanasundaram, and X. Zou, "Survey of return-oriented programming defense mechanisms," *Security and Communication Networks*, vol. 9, no. 10, pp. 1247–1265, 2016.

[18] T. Coudray, A. Fontaine, and P. Chifflier, "Picon: control flow integrity on llvm ir," in *Symposium sur la sécurité des technologies de l'information et des communications, Rennes, France*, 2015, pp. 3–5.

[19] R. Tschüter, J. Ziegenbalg, B. Wesarg, M. Weber, C. Herold, S. Döbel, and R. Brendel, "An llvm instrumentation plug-in for score-p," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, 2017, pp. 1–8.

[20] A. Krishnaswamy and R. Gupta, "Profile guided selection of arm and thumb instructions," *ACM SIGPLAN Notices*, vol. 37, no. 7, pp. 56–64, 2002.

[21] J. Stanier and D. Watson, "Intermediate representations in imperative compilers: A survey," *ACM Computing Surveys (CSUR)*, vol. 45, no. 3, pp. 1–27, 2013.

[22] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[23] H. Peeler, S. S. Li, A. N. Sloss, K. N. Reid, Y. Yuan, and W. Banzhaf, "Optimizing llvm pass sequences with shackleton: a linear genetic programming framework," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2022, pp. 578–581.

[24] G. B. Welearegai, C. Hu, and C. Hammer, "Detecting and preventing rop attacks using machine learning on arm," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2023, pp. 667–677.

[25] X. Huang, F. Yan, L. Zhang, and K. Wang, "Honeygadget: A deception based approach for detecting code reuse attacks," *Information Systems Frontiers*, vol. 23, pp. 269–283, 2021.

[26] F. Cassano, C. Bershatsky, and J. Ginesin, "Safellvm: Llvm without the rop gadgets!" *arXiv preprint arXiv:2305.06092*, 2023.

[27] B. Kim, K. Lee, W. Park, J. Cho, and B. Lee, "Rio: Return instruction obfuscation for bare-metal iot devices," *IEEE Access*, 2023.

[28] D. Eberius, T. Patinyasakdikul, and G. Bosilca, "Using software-based performance counters to expose low-level open mpi performance information," in *Proceedings of the 24th European MPI Users' Group Meeting*, 2017, pp. 1–8.