Faktor Exacta

Vol. 19, No. 1, March 2026, pp. 16~24
eISSN: 2502-339X, pISSN: 1979-276X, https://doi.org/10.30998/faktorexacta.v19i1.807

Bypassing CPU Protections using Function-Oriented Programming (FOP): A Linux Kernel Case Study

Suryo Bramasto[1], Muhammad Ramli[2]
[1,2]Department of Informatics, Institut Teknologi Indonesia, Indonesia

Article Info

Article history:

Received Sep 9, 2019
Revised May 20, 2020
Accepted Jun 11, 2020

Keywords:

Function-oriented programming
FOP Arbalest
Gadgets
Kernel
Linux
CPU Protections

ABSTRACT

Modern Central Processing Unit (CPU) hardware protections have significantly increased the difficulty of traditional code-reuse attacks, such as Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP). This study investigates the viability of Function-Oriented Programming (FOP), a novel exploitation technique, against systems implementing these modern architectural mitigations. Combining a comprehensive literature review with controlled simulations, this research introduces FOP Arbalest, an analytical artifact designed to identify, validate, and map exploitable FOP gadgets. Evaluation of the Linux kernel v5.19.17 demonstrates that FOP Arbalest identified 7,470, 18,541, and 52,285 gadgets at increasing analysis depths, yielding substantially more than conventional ROP and JOP techniques. Notably, FOP Arbalest is the first framework to integrate symbolic execution as the primary mechanism for FOP gadget detection, whereas prior x86-64 approaches relied exclusively on compiler plugins and static analysis. Future work will implement FOP techniques on Qualcomm Snapdragon system-on-chip (SoC) platforms to evaluate the portability and scalability of this vector across heterogeneous mobile computing environments.

Corresponding Author:

Suryo Bramasto,
Department of Informatics,
Institut Teknologi Indonesia,
Jl. Raya Puspiptek Serpong, Setu, Serpong, Tangerang Selatan.
Email: suryo.bramasto@iti.ac.id

1. INTRODUCTION

In Return-Oriented Programming (ROP), an attacker gains control of the call stack to hijack the program's control flow and subsequently executes a carefully selected sequence of machine instructions that already reside in memory, referred to as gadgets [1]. Each gadget typically terminates with a return instruction and resides within existing subroutines in the program code and/or linked libraries. When chained together, these gadgets enable an attacker to perform arbitrary computations on a target system, even in the presence of security defenses designed to mitigate code-reuse attacks [2]. Return-Oriented Programming (ROP) has been studied for more than fifteen years [3]. Since then, other code-reuse techniques have been shown to be effective under specific conditions, including Jump-Oriented Programming (JOP) [4]. As these techniques have evolved over the past decade, new defense mechanisms have been developed to limit their effectiveness. Among the most notable is Control-flow Enforcement Technology (CET) [5] for Intel processors, and Pointer Authentication (PAC) and Branch Target Identification (BTI) [6] for ARM processors. The objective of these protection mechanisms is to restrict the use of code-reuse attacks, such as ROP and JOP, against software applications.

BTI enforces specific landing targets for indirect branches, encompassing both register- and memory-based jump and call instructions. Typically placed at the entry of functions susceptible to indirect references, these landing pad directives serve as control-flow safeguards. In Intel processors, the designated landing pad

instruction is `ENDBR64`. Should an indirect jump or call fail to target this specific instruction, a trap is triggered. This mitigation mechanism aims to curtail the efficacy of JOP attacks. Although ROP and JOP have constituted the dominant paradigms in code-reuse attacks, the evolution of mitigations has advanced to a stage where hardware-based approaches represent the most effective mechanisms for fundamentally constraining these attack vectors.

The CET and PAC/BTI architectures are designed to restrict an attacker's ability to hijack execution control following memory corruption by impeding code-reuse attack techniques. Although the majority of memory corruption attacks leverage code-reuse techniques as their primary vector, constraining the scope of these methods has inadvertently introduced a disparity between perceived system confidence and the actual security posture achieved. The core problem investigated in this study is the identification of functional gaps in the implementation of CET and PAC/BTI that still permit the persistence of code-reuse attacks. Exploring these vulnerabilities aims to map the protective components not yet accommodated by existing defense mechanisms. By identifying and addressing these unprotected components, this research formulates potential solutions aligned with the original design objectives of these mitigation mechanisms.

With the adoption of CET and PAC/BTI architectures in next-generation systems, comprehensive validation has become imperative to ensure that the resource investment in designing and implementing these defense mechanisms yields measurable effectiveness, without overlooking edge cases or alternative exploitation scenarios. Mapping the gaps in existing defenses facilitates the development of targeted technical solutions that can be integrated into current systems as well as future architectures. This approach continuously enhances system security reliability by reducing reliance on implicit security assumptions. FOP enables code-reuse techniques to exploit previously undetected vulnerabilities. Consequently, this renders existing defense mechanisms ineffective, as the full capabilities of the exploitation techniques that were intended to be restricted can still be executed unhindered. FOP involves leveraging entire functions as gadgets. Unlike the traditional concept of a "gadget" in ROP or JOP scenarios, which typically comprises only a few instructions, such as the classic `POP RDI; RET;` sequence, FOP extends the definition of a gadget to encompass entire functions.

The objective of utilizing entire functions as gadgets is to exploit two inherent capabilities of a function. First, the ability to invoke a function for its intended purpose. For instance, by controlling the parameters of a function call such as `strcpy`, arbitrary memory manipulation becomes feasible; this principle can be extended to any function that includes the initial landing pad instruction in Glibc, such as `mprotect` or `syscall`. The second capability is derived from the side effects emerging from a function invocation. Because parameter registers are typically treated as volatile, restoring or preserving them is deemed unnecessary. Consequently, the values within these parameter registers remain potentially useful after the function returns.

The aforementioned capabilities rely on a reliable dispatcher gadget. In the previously cited example, the dispatcher must not alter the registers containing highly volatile arguments between successive FOP gadget invocations. This paper examines such dispatchers, which are typically found within Glibc and invoked during normal execution. Because dispatcher function calls do not require arguments, the parameter registers are not reset between invocations. This allows subsequent FOP gadgets to utilize the register parameters established by the preceding call. By leveraging a memory corruption vulnerability in conjunction with such a dispatcher, an FOP chain can be constructed to achieve arbitrary code execution.

FOP represents an evolution of the code-reuse attack paradigm, building upon the foundations of ROP and JOP. While mitigations such as Data Execution Prevention/No-Execute (DEP/NX) and Address Space Layout Randomization (ASLR) preclude the execution of injected code, and various other mechanisms constrain the viability of gadget-level chaining, FOP emerges as an approach that operates at the granularity of entire functions. Rather than chaining short instruction sequences (gadgets), FOP constructs malicious payloads by invoking existing functions within the target binary or shared libraries. By utilizing larger execution blocks and adhering to standard Application Binary Interface (ABI) conventions, including function prologues and epilogues, the manipulated function call sequences exhibit a more natural execution profile. Consequently, these sequences are significantly more resilient to detection by techniques that rely exclusively on identifying conventional gadget patterns [7].

The FOP process entails the identification of high-utility functions, such as those governing input/output (I/O) operations, memory allocation, or state modification, which, when orchestrated in sequence, can achieve objectives such as the exfiltration of sensitive data or the execution of specific actions. In contrast to ROP and JOP, which operate at the granularity of individual instructions, FOP necessitates mechanisms to coerce the program into invoking these functions in a predetermined sequence and with attacker-controlled arguments. This coercion is typically achieved through data structure corruption or control-flow manipulation that dictates the invocation context and timing of the targeted functions. Owing to its higher level of abstraction, FOP can evade detection mechanisms specifically engineered to identify gadget-level code reuse, thereby introducing novel challenges for mitigation systems that rely exclusively on the inspection of branch targets or instruction-level patterns.

Formally, the FOP attack class is defined through the categorization of function capabilities that are advantageous to an adversary, and it delineates theoretical or simulation-based scenarios wherein function chaining is utilized to achieve specific exploitation objectives. Furthermore, prior analyses have examined the impact of modern mitigations, including Control-Flow Integrity (CFI), shadow stacks, and micro architectural protections, on the feasibility of FOP, while concurrently identifying residual vulnerabilities within these defense mechanisms [8]. Furthermore, the evolution of detection tools and automated analysis techniques is imperative. Next-generation exploit detectors and generators must comprehend function semantics and intermediate representations (e.g., p-code/IR) to evaluate combinations of functions susceptible to abuse. FOP expands the code-reuse attack surface, as numerous programs encompass functions that, when orchestrated, can achieve complex objectives without necessitating gadget-level chaining [9]. Traditional mitigations effective against ROP and JOP may prove insufficient against FOP. For instance, CFI mechanisms that solely constrain control-flow targets continue to permit function call sequences that are syntactically legitimate yet semantically abused. Consequently, automated detection frameworks must evolve beyond instruction-pattern analysis to explicitly account for function invocation context and call semantics.

From a defensive standpoint, a comprehensive understanding of FOP is imperative for the design of more effective mitigation strategies. Proposed defensive approaches encompass the implementation of fine-grained, context-aware CFI mechanisms that verify not only call targets but also the invocation context and argument types; the deployment of semantic behavior monitoring to detect anomalous function call sequences; the restriction of sensitive Application Programing Interface (API) surfaces via the principle of least privilege and privilege separation; and the utilization of sandboxing alongside other isolation techniques to constrain the impact of potential function abuse [10]. Furthermore, the integration of static and dynamic analysis techniques that leverage code intermediate representations facilitates the identification of function combinations capable of executing malicious operations.

This study pursues two primary objectives. First, to evaluate the efficacy of FOP techniques within computing environments that implement state-of-the-art, CPU-based hardware mitigation mechanisms. Second, to employ a qualitative-analytical methodological framework, integrating a comprehensive literature review and controlled simulations, to develop a research artifact designated as FOP Arbalest. This analytical tool is engineered to facilitate the identification, classification, and validation of the functional utility of FOP gadgets possessing exploitation potential. The empirical fulfillment of these objectives demonstrates that FOP retains significant operational relevance within the contemporary security ecosystem, while yielding strategic implications for mapping the evolution of code-reuse attack vectors and informing the development of future adaptive mitigations.

The utility of FOP can be evaluated in terms of the access scope and computational capabilities afforded by this attack vector. In prior code-reuse attack exploitation literature [11], a fundamental parameter for evaluating the capacity of this technique is the demonstration of Turing completeness, defined as the capability to execute any algorithmically definable computable function within the available memory space. To establish FOP as a viable substitute for conventional exploitation techniques, this approach must demonstrate functional capabilities equivalent to those of both ROP and JOP. This study evaluates such equivalence through a systematic mapping of FOP utilization points, which are comparatively analyzed against ROP and JOP deployment scenarios within identical attack contexts parameter.

The principal results of this research encompass the implementation of FOP techniques and the development of a supporting tool for FOP gadget detection. Given that prior literature has established the feasibility of FOP under specific operational conditions, the focus of this study is redirected toward evaluating FOP deployment within environments incorporating state-of-the-art, CPU-based hardware mitigation mechanisms, alongside the introduction of the proposed FOP tool. From the perspective of scientific novelty, the FOP Arbalest architecture constitutes an original contribution to the domain of code-reuse attack research, demonstrating functional differentiation through the comprehensive fulfillment of all stipulated technical objectives

2. RESEARCH METHOD

The research environment for implementing FOP test cases employs a kernel-based setup representative of real-world scenarios. Specifically, the qemu-system-x86_64 emulator is utilized to replicate the execution of the Linux kernel image, version 5.19.17. The integration of this kernel image with a minimal BusyBox-based user-space environment facilitates the validation of exploits within an isolated and controlled kernel-space execution context. The scope of the evaluation is specifically concentrated on two hardware architectural platforms: the Intel Pentium N4200 (Goldmont microarchitecture) and the ARM Cortex-X2.

The research presented in this article commences by establishing the formal definition and threat model of FOP, thereby delineating its distinctions from ROP and JOP. Furthermore, it defines critical

environmental assumptions, including the availability of functions within target binaries and shared libraries, as well as the capability to manipulate data and data structures to influence function invocations [12]. Subsequently, an inventory and classification of function-level building blocks is conducted. This process entails identifying categories of functions advantageous to an adversary, specifically those governing I/O operations, memory allocation and manipulation, and process control, along with an analysis of the Application Binary Interface (ABI) and calling conventions. Furthermore, it examines the mechanisms by which function arguments can be controlled through data corruption or the manipulation of program structures [13].

To identify candidate functions and evaluate their viability for function chaining, this study employs an integrated approach combining static and dynamic analysis on a set of target binary samples. The research targets libc6 and the Linux kernel version 5.19.17, from which a total of 57,447 functions were extracted. The selection criteria encompassed presence within the export table, control-flow complexity or gadget depth, specifically 15, 25, and 50 nodes within the Control Flow Graph (CFG), and explicit involvement in memory management operations or privilege validation [14]. Static analysis was conducted using IDA 9.3 to perform disassembly, CFG reconstruction, symbol table mapping, and decomposition into pseudo-code/Intermediate Representation (IR), thereby facilitating the identification of parameter dependencies and function invocation patterns. Concurrently, dynamic analysis was implemented via runtime instrumentation leveraging angr 9.2.58 with the Unicorn backend. This configuration enabled real-time execution tracing, the observation of register state transitions, and the detection of side effects pertaining to memory allocation and the architectural state. All analytical outputs were processed automatically via a custom pipeline script to execute large-scale scanning and filter functions that satisfy the technical prerequisites for chaining. Final validation was conducted through the development of a Proof-of-Concept (PoC) and scenario simulation within the qemu-system-x86_64 environment, which empirically demonstrates the function chaining mechanism to achieve full privilege escalation [15].

During the implementation and evaluation phases of the artifact, the primary data objects automatically extracted constitute a catalog of identified FOP gadgets. This identification mechanism is designed to compile comprehensive technical metadata for each entry, encompassing the following critical attributes [16]:

1. Memory address: the specific location of the instruction within the kernel binary image.
2. Register impact: the modifications to flag bit states or operand values within architectural registers.
3. Memory read constraints: explicit conditions regarding memory location references that must be satisfied.
4. Memory writes constraints: the validity requirements for writing data to specific memory locations.
5. Branch prerequisites: Boolean logic conditions that must be fulfilled to enable control-flow transitions.
6. System call prerequisites: register argument conditions that must be satisfied to invoke the kernel function

The utilization of this data structure enables categorical analysis and quantitative evaluation of the operational utility of individual gadgets. This systematic approach facilitates the filtering and optimization of exploitation paths by constraining the search space to exclude non-viable gadgets. Furthermore, the extraction of these parameters provides transparency regarding the actual execution impact of a given gadget, thereby enabling the prediction of its behavioral consistency under runtime conditions. This analytical scope constitutes a direct manifestation of the symbolic execution principles underlying the gadget identification process, wherein the structured verification of critical points ensures the accuracy of data dependency mapping between registers and memory subsystems.

The data analysis in this study is predicated upon an evaluation framework comprising three primary methodological premises [17]. The first premise focuses on quantifying the functional utility of a gadget by computing the register state delta. This modification index is derived from a systematic comparison of register states prior to and following execution, serving as an indicator for identifying significant or unique state modifications. Algorithmically, code fragments that complete execution without altering register values is classified as No-Operation (NOP) instructions and are excluded by default from the FOP gadget candidate set. Nevertheless, in specific scenarios, NOPs may still be leveraged for padding or control-flow alignment. The second premise encompasses the verification of constraint feasibility. The application of symbolic execution facilitates the preliminary estimation of program paths and the requisite predicate conditions. However, empirical validation indicates that certain symbolically identified constraints may prove infeasible when subjected to specific memory limitations or runtime environmental conditions [18]. Consequently, this filtering stage is critical to ensure that only gadgets with computationally valid execution preconditions are retained.

The design of the FOP Arbalest artifact is extended through the integration of analytical constraints into the evaluation of conditional jumps. Within the symbolic execution framework, the processing of branch instructions is classified into three distinct execution states. The first state occurs when the branch condition is

deterministically verified as true, thereby definitively redirecting the control flow to the taken branch. The second state arises when the condition is proven false, resulting in a not-taken branch. The third state encompasses conditions that remain symbolic, wherein branch satisfiability depends on variable instantiation or environmental constraints that can only be resolved during actual execution scenarios. The implementation of this processing mechanism facilitates the simplification and functional validation of gadgets containing symbolic conditions, thereby enhancing their practical utility in exploit chain construction without compromising the accuracy of control-flow modelling.

This study further extends the register delta analysis to optimize gadget selection based on cross-architectural relevance. By constraining the search space to a subset of registers that play a crucial role in function calling conventions, specifically the Application Binary Interface (ABI), this methodology enables the selection of gadgets that maintain high utility across diverse platforms. As a comparative illustration, the RAX register in the x86-64 architecture and the X0 register in the ARM64 architecture both function as return value holders. However, under the ARM ABI standard (AAPCS64), X0 simultaneously serves as the carrier for the first argument in function calls. This contextual distinction enables researchers to deterministically filter gadgets that exclusively modify RAX without affecting argument registers, thereby enhancing the efficiency of exploit chain construction in ARM environments compared to generic approaches.

3. RESULTS AND DISCUSSION

A critical metric in evaluating the utility of FOP tooling is the execution time required to complete the analytical process. The data presented in Table 1 quantifies the computational time required to identify gadgets across three primary architectural categories utilizing the FOP Arbalest tool. These measurements are designed to provide an empirical assessment of the tool's operational efficiency in large-scale binary scanning scenarios, thereby facilitating a comparative analysis of processing overhead across diverse target environments. As theoretically predicted, a non-linear increase in execution time is observed as the quantity of identified gadgets escalates. This indicates that the growth in instruction depth does not exhibit a linear correlation with the number of successfully resolved test cases. Consequently, the computational complexity of the FOP Arbalest tool is not solely contingent upon the volume of gadgets; rather, it is fundamentally governed by the hierarchical structure and control-flow dependencies underlying the code fragment search process.

Table 1. Time Required for Gadget Identification

Lubrary	Gadget Depth		
	15	25	50
Libc(x64)	52,84 second	301,41 second	2484,6 second
Centos (x64)	31,74 second	255,95 second	2093,61 second
Fedora (x64)	50,74 second	281,52 second	1948,05 second
OpenSuse (x64)	31,70 second	247,79 second	1546,54 second
Ubuntu (x64)	52,69 second	298,18 second	1805,4 second

The target Linux kernel version for this study is 5.19.17 (released in November 2022), which provides comprehensive support for Control-flow Enforcement Technology (CET) implementation, including full Indirect Branch Tracking (IBT) support for landing pads, alongside novel protection mechanisms such as those applied to the prepare_kernel_cred function. The protection mechanisms introduced in kernel versions subsequent to 5.18 significantly constrain ROP and JOP attack chains by enforcing novel validation checks on functions that are frequently targeted by exploits. In any kernel-level attack, success is heavily contingent upon the availability of vulnerabilities that permit the exploitation of potential targets; notably, kernel versions post-5.18 incorporate additional mechanisms designed to remediate such vulnerability vectors. This study examines FOP primitives within the libc target in comparison to the Linux kernel. Given that the flexibility and diversity of primitives correlate directly with the number of discovered gadgets, a larger code base inherently implies an equivalent or greater magnitude of exploitable capabilities compared to libc primitives. Considering that the Linux kernel contains nearly ten times the number of gadgets compared to libc, this research confirms the availability of sufficient gadgets to demonstrate the same primitives identified in prior analyses. An additional critical observation is that, despite the increased volume of functions and potential gadgets, no identified gadgets facilitate the exploitation of values stored in the RAX register. This limitation persists across both user-space and kernel-space environments on Intel architectures. Figure 1 illustrates the finalized FOP chain deployed within the Linux kernel, which culminates in Linux privilege escalation.

Quantitatively, the total number of functions subjected to analytical verification within the target kernel vmlinux image is 57,447. This metric serves as an indicator of analysis coverage, facilitating a

comparative evaluation of the FOP Arbalest tool's scalability when processing operating system-scale binary artifacts versus user-space libraries. The substantial increase in function quantity within the Linux kernel implies a corresponding expansion of the search space for potential gadget identification, thereby directly escalating the temporal complexity of the analytical process. Table 2 presents the number of gadgets successfully identified within the target kernel vmlinux image, serving as an empirical representation of the tool's detection capabilities in high code-density environments. The extensive code base of the Linux kernel directly necessitates a broadened search space for gadget detection. Consequently, the volume of identifiable potential dispatcher gadgets is proportionally larger, yielding 308 gadgets for Linux kernel version 5.19.17, in stark contrast to environments with constrained code coverage.

Table 2. Number of Gadgets per Gadget Depth

Gadget Depth	Gadgets Count
15	7470
25	18541
50	52285

This vulnerability enables memory corruption via a heap overflow, wherein the attacker can manipulate the msg_msg structure, leading to the leakage of the Linux kernel and kernel heap addresses, thereby bypassing Kernel Address Space Layout Randomization (KASLR). This subsequently permits the overwriting of the pipe_buffer structure along with its _ops pointer, resulting in an arbitrary function invocation with the RSI register pointing to attacker-controlled memory. The initial Proof-of-Concept (PoC) employed this technique to perform a stack pivot to RSI and initiate an ROP chain. In this FOP instance, however, the pivot is directed to the FOP dispatcher. A challenge encountered during testing was the absence of identified FOP dispatchers originating from the RSI register. To circumvent this issue, an RSI-to-RDI pivot gadget, as illustrated in Figure 2, is required, which facilitates the utilization of the dispatcher depicted in Figure 1.

Once the dispatcher is invoked, the standard kernel privilege escalation process commences. The demonstrated FOP chain leverages the fact that controlling the first argument in a call to prepare_kernel_cred with a null value returns a copy of the kernel credential structure. No identified gadget possesses the capability to utilize the value stored in the RAX register. Consequently, the kernel credential structure generated by prepare_kernel_cred and returned in RAX is rendered unusable. However, prepare_kernel_cred exhibits a side effect of storing the init_cred structure in the RSI register upon function return. Transferring this pointer to the RDI register and subsequently invoking commit_cred enables the FOP chain to elevate the current privileges to root, thereby successfully completing the privilege escalation.

Figure 2 represents the output of the symbolic execution process, which parses the register state and memory access traces at a specific address within the x86-64 kernel space, specifically at location 0xffffffff813aaf00. In the Results section, the RDI register status is expressed symbolically as 0xffffffffffffffc8 + [40 + RSI], indicating an arithmetic operation between a negative offset (in two's complement representation) and a value dereferenced from memory. This memory value itself is computed based on the summation of the constant 40 and the contents of the RSI register. Furthermore, the Symbolic Target line displays the expression [16 + [0x78 + [40 + RSI]]], which represents a chain of nested pointer indirection spanning three levels of dereferencing. Technically, this computational structure confirms the existence of gadgets possessing the capability to manipulate pointer references recursively. In the context of vulnerability analysis, such memory access patterns signify potential FOP attack vectors that exploit dynamic data dependencies to construct exploit chains capable of evading detection by conventional architectural mitigation mechanisms.

The final capability engineered is the successful return to user space with elevated privileges. The conventional ROP approach for returning to user space entails saving the execution state onto a controlled stack and leveraging a kernel mechanism designed for the transition back to user space. However, to maintain compliance with the shadow stack specifications established herein, no stack modifications can occur during the attack, rendering this approach unfeasible. Consequently, the FOP chain can employ two alternative approaches to return to user space. The first approach involves designing a dispatcher along the FOP chain to safely exit once the privilege escalation is complete. This is viable because the kernel assumes normal execution flow, given that no modifications have been made to the stack or memory space. Nevertheless, this method becomes unfeasible because the utilized dispatcher employs the R12 register as a counting mechanism. This limitation is determined empirically through testing and by analyzing the dispatcher depicted in Figure 1, which reveals that the R12 register is derived from the RDI and RSI registers. In this context, the RSI and RDI registers within the dispatcher point to attacker-controlled memory, causing R12 to assume an unrealistically large value for iteration. The second approach involves leveraging the telefork technique.

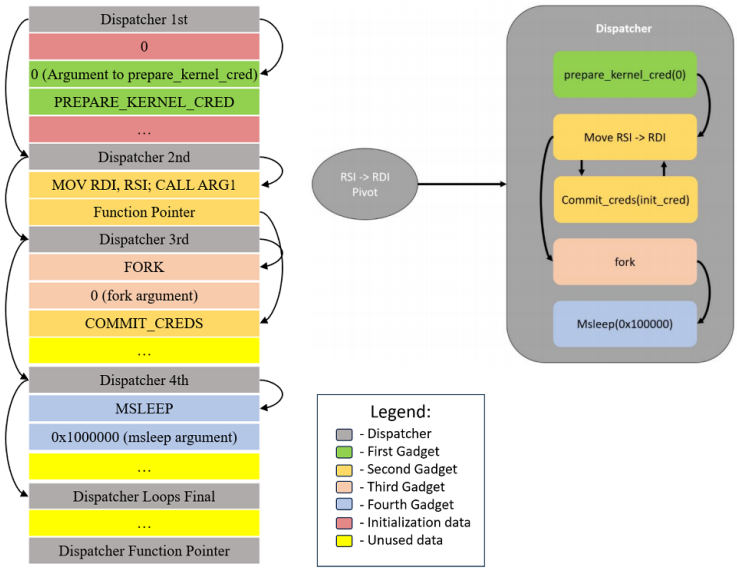

Figure 1. FOP kernel Chain in memory

```
KERNEL 0xffffffff813aaf00:
   Results:
      ...
      RDI: 0xffffffffffffffc8 + [40 + RSI]
      ...
   Symbolic Target:
            [16 + [0x78 + [40 + RSI]]]
```

Figure 2. RSI-to-RDI pivot gadget

The telefork technique operates by invoking the fork system call from within the kernel. Subsequently, this action spawns a secondary thread within the user-space application precisely at the execution point where the exploit initially halted. Concurrently, the original thread remains within the Linux kernel and is configured to wait indefinitely via the msleep function. This configuration causes the kernel thread to become permanently blocked, thereby creating the illusion of normal execution from an external perspective. Given that this approach merely constitutes a standard function invocation, it represents a prime target for FOP exploitation. This final mechanism enables the FOP attack to successfully transition back to user space, culminating in a successful Linux privilege escalation achieved exclusively through the utilization of FOP gadgets. The demonstration results confirm that FOP can successfully execute full privilege escalation within the Linux kernel, even in the presence of CET, CFI, and shadow stacks, provided that the attacker can control the data flow and leverage available dispatchers. Table 3 presents the number of gadgets that can be utilized in conjunction with the dispatcher to achieve full privilege escalation in the Linux kernel across respective code-reuse attack paradigms.

Tabel 3. Perbandingan FOP, ROP, dan JOP

Kedalaman *Gadget*	Jumlah *Gadget* Teridentifikasi pada *Kernel* Linux versi 5.19.17 dan *Processor* x64		
	FOP	ROP	JOP
15	7470	370	86
25	18541	835	226
50	52285	1261	448

4. CONCLUSION

The protection mechanisms of CET and PAC/BTI are designed to constrain the efficacy of code-reuse attacks. Demonstrating the capabilities of FOP techniques within environments that have adopted these defense mechanisms can expose functional gaps or limitations in the validation coverage of existing security implementations. These empirical results indicate that FOP possesses the potential to achieve exploitation

capabilities equivalent to those of other code-reuse attack vectors explicitly targeted by CET and PAC/BTI mitigations. By substantiating the continued viability of FOP techniques in protected environments, mapping the vulnerabilities within these defense mechanisms facilitates iterative design processes and the refinement of mitigation policies. Consequently, the fundamental objective of restricting code-reuse attacks can be realized more comprehensively and with greater resilience against the evolution of exploitation techniques.

As is customary with code-reuse techniques, the execution of FOP-based attacks fundamentally depends on the availability of exploitable gadgets. Consequently, the identification of FOP gadgets constitutes an imperative technical prerequisite for the successful realization of this attack vector. Within the contemporary landscape of detection methodologies, FOP Arbalest represents the pioneering implementation that integrates symbolic execution as the primary mechanism for FOP gadget detection. Previously, approaches available for the x86-64 architecture environment relied exclusively on compiler plugins coupled with static analysis. The adoption of symbolic execution in FOP Arbalest introduces a significant comparative advantage: temporally, it accelerates the identification process compared to conventional static analysis methods; technically, it enables the detection of gadgets exhibiting higher control-flow complexity and intricate data dependencies, thereby overcoming the inherent limitations of compiler plugin-based approaches.

The development of FOP Arbalest adheres to a qualitative-analytical methodological framework, incorporating a comprehensive literature review and controlled simulations to yield a research artifact. This process systematically involves the development and validation of the artifact to address identified problems. This approach facilitates the integration of empirical and qualitative methods within the artifact evaluation cycle, thereby enabling comprehensive performance measurement and feasibility analysis. Furthermore, the design theory underpinning the artifact's development serves as a conceptual framework that elucidates the mechanisms for fulfilling both functional and non-functional requirements, ensuring that the resulting solution aligns with operational necessities and existing architectural constraints.

The demonstration of FOP attacks on the Linux kernel substantiates that this technique can operate effectively within modern environments equipped with state-of-the-art CPU mitigations. FOP expands the code-reuse attack surface by operating at the function level of abstraction, thereby evading detection mechanisms that focus exclusively on instruction patterns or branch targets. This study underscores that mitigations such as CET, CFI, and shadow stacks must be augmented with call context validation, function semantic monitoring, and the restriction of critical API surfaces in accordance with the principle of least privilege. The development of next-generation detection tools must integrate static and dynamic analysis based on IR to map combinations of functions susceptible to abuse. Future research is recommended to explore the automation of dispatcher discovery, the evaluation of FOP on the ARM64 architecture with PAC/BTI, and the development of control-flow graph-based mitigations that incorporate call semantics.

The findings of this study reinforce the epistemological validity of the adopted qualitative-analytical methodological framework, which integrates a literature review and controlled simulations to yield a research artifact, through empirically verified substantive contributions. This approach necessitates clear, measurable research outputs with verifiable relevance to the target application domain. The implementation of the FOP Arbalest module not only satisfies these ideal criteria but also comprehensively demonstrates the operationalization of the aforementioned methodological principles within the context of system security engineering. Furthermore, this research introduces an original methodological approach to defining the application scope of FOP techniques by establishing state-of-the-art, CPU-based protection mechanisms as a previously unexplored design target. The contribution to novelty is further extended through the successful implementation of FOP on the Linux kernel, marking a first in the academic literature concerning code-reuse attacks. The accumulation of these factors consolidates the methodological foundation and strategic direction of this study. As the subsequent iteration of development, the research roadmap encompasses the implementation of FOP techniques on Qualcomm Snapdragon system-on-chip (SoC) platforms to validate the portability and scalability of this attack vector within a heterogeneous mobile device ecosystem.

ACKNOWLEDGEMENT

This research was supported by the Institute of Teknologi Indonesia for the 2025/2026 academic year.

DAFTAR PUSTAKA

[1] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, “PACMAN,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, New York, NY, USA: ACM, Jun. 2022, pp. 685–698. doi: 10.1145/3470496.3527429.

[2] K. Sun, C. Hu, Y. Zhuang, and J. Wang, “Code Reuse Attack Detection Based on Hardware and Software Performance Counters,” in *2025 5th International Conference on Consumer Electronics and*

Computer Engineering (ICCECE), IEEE, Feb. 2025, pp. 313–317. doi: 10.1109/ICCECE65250.2025.10985204.

[3] S. Ognawala, F. Kilger, and A. Pretschner, "Compositional Fuzzing Aided by Targeted Symbolic Execution," Oct. 2019.

[4] R. C. Goluch, "Trust, transforms, and control flow: A graph-theoretic method to verifying source and binary control flow equivalence," Iowa State University, 2021. doi: 10.31274/etd-20210609-59.

[5] M. Lipp *et al.*, "Meltdown," *Commun. ACM*, vol. 63, no. 6, pp. 46–56, May 2020, doi: 10.1145/3357033.

[6] S. Matsuoka, "Fugaku and A64FX: the First Exascale Supercomputer and its Innovative Arm CPU," in *2021 Symposium on VLSI Circuits*, IEEE, Jun. 2021, pp. 1–3. doi: 10.23919/VLSICircuits52068.2021.9492415.

[7] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A Survey for Roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–36, Jan. 2022, doi: 10.1145/3512345.

[8] D. Reid, M. Jahanshahi, and A. Mockus, "The extent of orphan vulnerabilities from code reuse in open source software," in *Proceedings of the 44th International Conference on Software Engineering*, New York, NY, USA: ACM, May 2022, pp. 2104–2115. doi: 10.1145/3510003.3510216.

[9] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E Platform," *ACM Transactions on Computer Systems*, vol. 30, no. 1, pp. 1–49, Feb. 2012, doi: 10.1145/2110356.2110358.

[10] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks," in *Proceedings 2014 Network and Distributed System Security Symposium*, Reston, VA: Internet Society, 2014. doi: 10.14722/ndss.2014.23156.

[11] K. Lin, H. Xia, K. Zhang, and B. Tu, "AddrArmor: An Address-based Runtime Code-reuse Attack Mitigation for Shared Objects at the Binary-level," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, IEEE, Sep. 2021, pp. 117–124. doi: 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00029.

[12] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, May 2019, doi: 10.1145/3182657.

[13] S. Ozdemir, R. Saptarshi, A. Prakash, and D. Ponomarev, "Track Conventions, Not Attack Signatures: Fortifying X86 ABI and System Call Interfaces to Mitigate Code Reuse Attacks," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, IEEE, Sep. 2021, pp. 176–188. doi: 10.1109/SEED51797.2021.00029.

[14] D. Parygina, A. Vishnyakov, and A. Fedotov, "Strong Optimistic Solving for Dynamic Symbolic Execution," in *2022 Ivannikov Memorial Workshop (IVMEM)*, IEEE, Sep. 2022, pp. 43–53. doi: 10.1109/IVMEM57067.2022.9983965.

[15] S. Xu, P. Xie, and Y. Wang, "AT-ROP: Using static analysis and binary patch technology to defend against ROP attacks based on return instruction," in *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, IEEE, Dec. 2020, pp. 209–216. doi: 10.1109/TASE49443.2020.00036.

[16] Y. Huang, F. Xu, H. Zhou, X. Chen, X. Zhou, and T. Wang, "Towards exploring the code reuse from stack overflow during software development," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, New York, NY, USA: ACM, May 2022, pp. 548–559. doi: 10.1145/3524610.3527923.

[17] B. He, Y. Guo, H. Liang, Q. Wang, and G. Xie, "Research on Defending Code Reuse Attack Based on Binary Rewriting," in *2022 IEEE 8th International Conference on Computer and Communications (ICCC)*, IEEE, Dec. 2022, pp. 1682–1686. doi: 10.1109/ICCC56324.2022.10065883.

[18] Z. Tianning, C. Miao, Z. Diming, and H. Hao, "Timing and Speculative Execution Attacks: Defeating State-of-the-Art Code-Reuse Defenses," *IEEE Access*, vol. 13, pp. 93084–93101, 2025, doi: 10.1109/ACCESS.2025.3573038.