SACK: Systematic Generation of Function Substitution Attacks against Control-flow Integrity

Zhechang Zhang*, Hengkai Ye*, Song Liu†‡ and Hong Hu*
*The Pennsylvania State University, †University of Delaware
{zhechang, hengkai, honghu}@psu.edu, songliu@udel.edu

***Abstract*—Control-flow integrity (CFI) is a widely adopted defense against control-flow hijacking attacks, designed to restrict indirect control transfers to a set of legitimate targets. However, even under a precise static CFI policy, attackers can still hijack control flow through function substitution attacks (SUB attacks), by replacing one valid target with another that remains within the allowed set. While prior work has demonstrated the feasibility of such attacks through manual construction, no approach constructs them systematically, scalably, and in an end-to-end manner.**

In this work, we present SACK, the first systematic framework for automatically constructing SUB attacks at scale. SACK collects triggered indirect call targets from benign executions and synthesizes security oracles with the assistance of a large language model. It then automatically performs target substitutions and leverages security oracles to detect security violations, while ensuring that execution strictly adheres to precise CFI policies. We apply SACK to seven widely used applications and successfully construct 419 SUB attacks that compromise critical security features. We further develop five end-to-end exploits based on historical bugs in SQLite3, V8 and Nginx, enabling arbitrary command execution or authentication bypass. Our results demonstrate that SACK provides a scalable and automated pipeline capable of uncovering large numbers of end-to-end attacks across diverse applications.

I. INTRODUCTION

Control-flow integrity (CFI) is a principled defense mechanism against control-flow hijacking attacks [1], [9]. These attacks exploit memory-safety issues, such as buffer overflow and use-after-free, to corrupt function pointers or return addresses, and divert the control flow to attacker-crafted logic [73], [7], [75], [12], [64]. To prevent malicious indirect control-flow transfers (ICTs), CFI techniques first identify a set of valid targets for each ICT instruction (*e.g.*, indirect call and indirect jump), and then enforce at runtime that all ICT transfers remain within this set [91], [80], [67]. A wide body of research has focused on refining these valid targets to improve both security and compatibility, like type analysis [80], [66], [83], [57], [60], [84], [10] and data-flow analysis [50], [26], [47], [51]. Recent advancements integrate hybrid analysis [58], leverage runtime context [81], [67], [39] and incorporate machine learning and large language models [92], [18] to further enhance precision.

Despite these advancements, researchers have identified a fundamental limitation: even a perfectly precise CFI policy cannot block all control-flow hijacking attacks. A seminal work by Carlini *et al.*, known as control-flow bending (CFB) [11], exposes this weakness. CFB shows that attackers can bypass CFI by substituting one valid target for another, both permitted under a fully precise static policy where each target is reachable by some benign input. Using this strategy, CFB achieves severe consequences, including arbitrary code execution and file modification, illustrating worst-case scenarios even under strong CFI defenses. Since CFB combines function pointer substitution with other exploitation techniques, like data-only attacks [16], [40], [44], we use the term *function substitution* attacks, or SUB attacks, to specifically describe attacks that modify function pointers between CFI-allowed targets.

However, while CFB demonstrates the feasibility of SUB attacks, it relies on manual construction and presents only two concrete examples. As a result, it remains unclear whether such attacks are practically achievable at scale or applicable to a broader range of real-world programs. In other words, we lack a systematic method for generating them and a clear understanding of their prevalence in modern software. The central focus of this work is to address this fundamental gap.

A straightforward approach to generating SUB attacks is to substitute the target of each ICT instruction with every CFI-allowed alternative and evaluate the resulting security consequences. However, this method faces two fundamental challenges. First, in the absence of a perfectly precise static analysis [26], it is difficult to determine the exact set of valid targets. Existing CFI solutions intentionally over-approximate these sets to preserve program functionality, which introduces many false targets [60], [51], [52]. SUB attacks, however, must avoid such spurious targets as any redirection to an invalid target would be blocked by an idealized CFI policy. Second, no standard mechanism can assess the security impact of function substitutions. Traditional control-flow attacks typically aim for arbitrary code execution [4], [75], [74], but SUB attacks are constrained to legitimate CFI-compliant targets. Their impacts depend on the high-level semantics of both the original and substituted functions. Focusing solely on code execution may miss subtle but critical consequences, such as bypassing authentication, that reflect the real-world danger of SUB attacks.

To address these challenges, we introduce SACK, the first system that systematically and at scale automates the construction of function substitution (SUB) attacks against precise

‡Song was affiliated with Penn State at the time of this work.

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.242317
www.ndss-symposium.org

control-flow integrity (CFI) in general-purpose applications. First, to overcome the lack of a complete set of allowed targets, SACK dynamically monitors program executions using available benign inputs to collect a ground-truth subset of valid targets. Any sound and precise CFI policy must allow these observed targets to preserve normal program behavior [11]. Second, to evaluate the security consequences of substitutions, SACK introduces security oracles, each consisting of concrete program inputs and expected security behaviors. Any deviation from this behavior after a substitution indicates a potential semantic violation, including subtle forms of security bypass. This approach enables SACK to capture a broader range of real-world impacts beyond traditional code execution. To scale oracle construction and reduce manual effort, SACK focuses on commonly implemented security features, such as authentication, rate limiting and audit logging, and utilizes large language models (LLMs) to automatically extract and synthesize behavior specifications from program documentation. This LLM-assisted design enables SACK to generalize across diverse applications, even in the absence of formal specifications.

It is important to note that the goal of this work is to demonstrate the practicality of constructing SUB attacks, rather than to exhaustively explore all security features and inputs. Therefore, we target representative security mechanisms and use curated inputs that exercise these features. This design allows us to highlight the feasibility and impact of SUB attacks under realistic conditions. We leave the broader exploration of security feature sets and input diversity to future work.

We implement the framework to systematically construct SUB attacks using 1,418 lines of C/C++ code and 300 lines of Python. First, SACK leverages program documentation and a large language model to generate a set of security oracles, which define expected security behaviors under specific inputs. Second, it executes the program using a set of test cases to collect dynamically observed indirect call targets. Finally, SACK re-executes the program with each input, and during each re-execution, it substitutes the target of one indirect call with a distinct alternative from the collected target set. It applies the corresponding oracle to assess whether the substitution causes the program to violate the intended security behavior. If so, SACK reports the case as a potential SUB attack. To improve the performance, SACK incorporates optimization techniques from the fuzzing community [89], [30], [59], [36], such as fork-server execution [88], to increase execution throughput and maximize the chances of constructing SUB attacks.

We apply SACK to seven real-world programs commonly targeted by control-flow hijacking attacks, like web servers, file servers, a database management system, and a JavaScript engine. SACK successfully constructs 419 SUB attacks that break 18 critical security features. These attacks bypass authentication, enable arbitrary code execution, disable security logging, violate resource limits, or break sandbox mechanisms. We also construct five end-to-end exploits using historical vulnerabilities. To our knowledge, this is the first work to demonstrate large-scale, automated construction of SUB attacks. Notably, SACK does not require deep manual understanding of the program source code and avoids heavyweight program analysis for identifying ICT targets. These results demonstrate that SACK provides a scalable and automated pipeline for constructing large numbers of CFI-bypassing SUB attacks across diverse, security-critical, real-world applications.

```
1  int sqlite3AuthCheck(Parse *pParse, ...)
2    sqlite3 *db = pParse->db;
3    if (db->xAuth == 0) return SQLITE_OK;
4    return db->xAuth(db->pAuthArg, ...); // indirect function call
5
6  int shellAuth(void *pClientData, ...) //--------- valid target 1
7    if (p->bSafeMode) safeModeAuth(pClientData, ...);
8    return SQLITE_OK;
9
10 int safeModeAuth(void *pClientData, ...) //------ valid target 2
11   if (... /* command is not allowed */)
12     exit(1);
13   return SQLITE_OK;
14
15 int idxAuthCallback(void *pClientData, ...) //--- valid target 3
16   idxMalloc(&rc, sizeof(IdxWrite));
17   return rc;
```

Fig. 1: Motivating example from SQLite3. Line 4 is the indirect function call, which has three valid targets: `shellAuth`, `safeModeAuth` and `idxAuthCallback`. We simplify the code for the sake of readability.

In summary, our work makes the following contributions.

- We propose a systematic approach that combines dynamic indirect-call target collection with LLM-assisted oracle generation to automate end-to-end function substitution attacks.
- We develop SACK, the first scalable framework that automatically modifies control-flow data to generate SUB attacks while remaining compliant with perfectly precise CFI policies.
- We apply SACK on 22 security features across seven widely used applications. It successfully constructs 419 SUB attacks, revealing the strong practicality of automating these attacks at scale bypassing precise CFI protections.

Open source. The source code of SACK and necessary instructions for reproducing the results are available at https://github.com/psu-security-universe/sack.

II. Background

A. The Motivating Example

Figure 1 presents the motivating example we extract from SQLite3, a widely used database management system. SQLite3 supports running SQL statements in a *safe* mode, which restricts database operations to a limited set of trusted SQL functions and extensions, preventing execution of potentially dangerous or untrusted code [79]. Function `sqlite3AuthCheck` conducts an authorization check to determine whether an action embedded in the `pParse` is allowed. If so, it returns `SQLITE_OK` so that the action will be executed; otherwise, it returns `SQLITE_DENY`, which rejects the action. Line 3 checks whether the authentication handler is `NULL`. If no, it will invoke the handler indirectly via the function pointer `xAuth` to make a decision (line 4). The indirect function call has three valid targets. `shellAuth` does not check anything in the normal mode, and invokes `safeModeAuth` in the safe mode; it is a dummy implementation and always returns `SQLITE_OK`. `safeModeAuth` compares the action with a predefined list of prohibited functions (line 11, details skipped), such as `load_extension` and writefile, and exits the process if the action is prohibited (line 12). idxAuthCallback merely checks whether the process has enough memory to create an index (line 16); it is mainly used when SQLite3 creates or uses index-based expressions.

When we run SQLite3 in safe mode, the indirect call at line 4 will invoke safeModeAuth to conduct necessary checks.

B. Control-flow Hijacking and Control-flow Integrity

Suppose attackers identify a memory-safety vulnerability in SQLite3 that enables arbitrary memory writes. They can overwrite the function pointer db->xAuth with the address of any executable code. This includes security-critical functions such as system for ret2libc attacks [64], or carefully crafted instruction sequences for return-oriented programming (ROP) [75], [7], [73]). When the indirect cal is executed, the control flow is hijacked, allowing attackers to execute arbitrary code. These are known as control-flow hijacking attacks.

To prevent such attacks, researchers develop control-flow integrity (CFI) [1] to protect indirect control-flow transfers (ICTs). The core idea is to infer a set of allowed targets for each ICT instruction, and perform runtime checks to ensure that all transfers remain within this set. Any deviation is treated as an attack and will terminate the execution. In Figure 1, a CFI technique may infer that the allowed targets for line 4 are shellAuth, safeModeAuth, and idxAuthCallback. If an attacker corrupts db->xAuth to a disallowed function, CFI will detect the mismatch and stop the execution to prevent damages.

The strength of a CFI mechanism depends on the precision of its target inference algorithm, where more precise analysis yields fewer allowed targets and stronger protection. To this end, prior work has developed various algorithms to refine target sets, including type analysis [80], [66], [83], [37], [57], multi-layer type analysis [60], [84], [10] and data-flow analysis [50], [26], [47], [51]. Recent work integrates multi-dimensional analysis [58], utilizes runtime information [81], [67], [39], and adopts machine learning [92] and large language models [18] to further improve precision. With these efforts, CFI has been deployed in real-world critical systems [34], [62], [35], [70].

C. Attacks against Perfectly Precise Static CFI

Despite recent improvements, researchers have uncovered generic strategies to bypass CFI [12], [21], [33], [26], [11]. Among these efforts, Carlini *et al.* propose control-flow bending (CFB) [11], a technique that bypasses even a perfectly precise static CFI policy. A CFI policy is considered perfectly precise if every allowed target for each ICT instruction can be triggered by some benign input. The resulting control-flow graph (CFG) is minimal since removing any edge would break legitimate functionality. Since practical CFI policies must preserve correct behaviors, they cannot restrict targets beyond this point. CFB demonstrates that even under this strong assumption, attackers can substitute function pointers from one valid target to another within the allowed set. In their study, Carlini *et al.* manually construct two such attacks: one achieving arbitrary code execution in Apache, and another enabling file manipulation in Wireshark. Since their work encompasses both control-flow and data-only attacks, we refer to the variant involving function-pointer corruption as *function substitution* (SUB) attacks.

Building on this concept, we construct a SUB attack on SQLite3 using the related code in Figure 1. When SQLite3 runs in safe mode, the indirect call at line 4 invokes safeModeAuth to block dangerous functions. However, by exploiting a vulnerability like CVE-2017-6983, an attacker can overwrite the function pointer db->xAuth with another allowed target, idxAuthCallback. idxAuthCallback merely checks memory availability for index creation and likely returns SQLITE_OK even for dangerous operations. As a result, the attacker bypasses safe mode, and invokes a prohibited function, like using load_extension to load and execute malicious shared libraries.

III. PROBLEM & CHALLENGES

Problem statement. Despite the feasibility demonstrated by Carlini *et al.*, function substitution (SUB) attacks have only been explored through isolated, manually constructed cases. No prior work provides a systematic or automated method for generating such attacks, leaving open questions about their practicality and impact. This paper addresses this gap by introducing the first automated framework for generating and evaluating SUB attacks across real-world general-purpose applications.

Given the nature of SUB attacks, a straightforward strategy for constructing them is to systematically substitute the target of each indirect control-transfer (ICT) instruction with another allowed target, and evaluate the resulting security impact. By exhaustively testing these substitutions and monitoring for deviations in expected behavior, we can assess the feasibility and prevalence of SUB attacks. In the case of Figure 1 where SQLite3 runs in safe mode, we can execute the program three times, each time forcing the indirect call to invoke a different legitimate target. When safeModeAuth is substituted with idxAuthCallback, a prohibited action unexpectedly succeeds, indicating a successful bypass of the safe mode.

A. Challenges

However, the straightforward method faces two key challenges. (**C1**) Constructing SUB attacks requires precise knowledge of all valid targets for each ICT instruction under precise CFI policies. However, no existing static analysis can provide such precision [26]. Most CFI implementations intentionally over-approximate target sets to preserve program functionality [51], [60], which introduces false targets unsuitable for SUB attacks. Instead, SUB attacks must remain valid under future precise CFI enforcement, so under-approximation is preferred. For instance, in Figure 1, identifying only safeModeAuth and idxAuthCallback (and omitting shellAuth) suffices to build the attack, while including disallowed targets like system would result in an attack that a precise CFI policy will reject. (**C2**) Assessing the security impact of a substitution is non-trivial. Unlike classic control-flow hijacking attacks that aim for arbitrary code execution [4], [74], SUB attacks may yield subtler consequences, such as bypassing authentication or disabling input sanitization. These impacts, while less extreme, can still enable severe exploits, as shown in data-only attacks [16], [40], [44]. However, detecting such effects requires program-specific, security feature-oriented oracles capable of identifying when critical protections are violated. Constructing these oracles is challenging, and typically requires substantial manual analysis to understand the program's high-level semantics, configuration, and expected behavior. For example, the SUB attack in §II-C requires understanding that SQLite3's safe mode is enabled via the `-safe` flag, that it prohibits certain actions like `load_extension`, and what behavior indicates that the restriction was bypassed. This level of insight is essential but difficult to obtain automatically.

B. Threat Model

We adopt a widely accepted threat model, consistent with prior work on control-flow attack and defense [11], [26], [82], [75], [52], [9]. The target is a benign application that contains memory-safety vulnerabilities but no malicious logic. The attacker can exploit these vulnerabilities to achieve arbitrary memory reads and writes. The application is protected by standard defenses, including W$\oplus$X [61] and shadow stacks [55], [43], [20]. Given well-known limitations [6], the attacker can bypass ASLR [71] using common techniques [41], [27], [45], [68], [63]. The assumption of arbitrary memory access is a commonly adopted abstraction to isolate the effects of memory manipulation from the specific techniques to obtain it. This allows us to focus on the construction of SUB attacks once such access is available. In practice, SUB attacks may succeed with more constrained attacker capabilities [56], [95], [90].

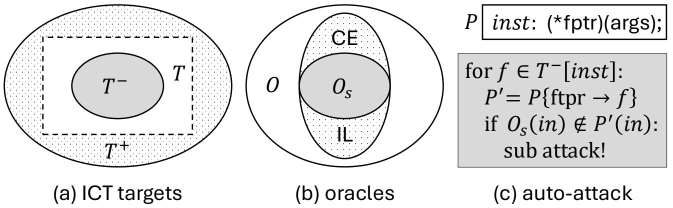
We further assume the presence of a fully precise static CFI mechanism. Although such precision is unattainable in practice, it models the strongest protection that static CFI could possibly provide. To simplify the design and implementation of our work, we assume the availability of the program's source code. However, this is not a fundamental limitation, and our work can be extended to support building attacks in COTS binaries.

IV. METHOD OVERVIEW

To address the aforementioned challenges, we propose three core techniques for systematically constructing SUB attacks in general-purpose applications. Figure 2 presents an overview of our approach. First, we define the notion of *sub-ground truth*, the set of ICT targets dynamically observed during benign executions, to ensure that each substitution remains valid under any precise CFI policy. Second, we introduce the concept of security-oriented oracles to evaluate the impact of each substitution. These lightweight oracles are derived from high-level security features specified in program documentation. Third, we develop an automated technique that substitutes ICT targets with entries from the sub-ground truth and verifies the resulting behavior using the constructed oracles.

A. Sub-ground Truth of ICT Targets

We address challenge **C1** by collecting the targets of indirect calls observed during dynamic executions with benign test cases. We refer to this conservative approximation as the *sub-ground truth*. Figure 2(a) demonstrates the relationship among three relevant target sets. Let T denote the theoretical ground truth, *i.e.*, the set of valid targets allowed by a fully precise static CFI policy. Although recent techniques have made progress toward approximating T [51], [58], [84], [47], no practical method can infer it with full precision. Current static analysis techniques instead produce a superset T^+ [18], [58], [51], which must include all targets in T to preserve program correctness. However, T^+ often contains false positives, *i.e.*, targets not actually allowed under a precise CFI policy, making them unsuitable for constructing SUB attacks. In contrast, we define T^- as the set of ICT targets observed during dynamic execution with benign inputs (*i.e.*, executions that do not trigger memory errors). As previously established [11], all elements in T^- are valid targets under T, ensuring that any substitutions confined to T^- respect even the strictest static CFI policy.



(a) ICT targets (b) oracles (c) auto-attack

Fig. 2: Method overview. Given a program P, we collect triggered targets T^- at runtime, and construct security-oriented oracles O_s. Then, we substitute each target and identify SUB attacks via oracles.

To maximize SUB attack success, we utilize available test cases to trigger as many ICT instructions as possible. Many programs ship with built-in test suites for functional validation and bug detection [77], [65], [2], offering a practical foundation for dynamic target collection. While full coverage remains challenging, this limitation is acceptable for us since our goal is to demonstrate the practicality rather than completeness. A substantial number of successful SUB attacks across programs is sufficient to validate feasibility. Moreover, recent advances in program-testing techniques, such as fuzzing [89], [8], [93] and symbolic execution [13], [14], [72], have significantly improved code coverage and can be incorporated into future extensions of our framework to further enhance target discovery.

In our motivating example (Figure 1), SQLite3 developers provide an extensive test suite [77] in the source repository and official release. Simply running the built-in command `make test` triggers two out of three legitimate targets for the indirect call at line 4, in particular, `safeModeAuth` and `idxAuthCallback`. By substituting `safeModeAuth` with `idxAuthCallback`, attackers can bypass safe mode restrictions.

B. Behavioral Oracles of Security Features

To address challenge **C2**, we propose constructing behavioral oracles for program security features. These oracles capture the expected behaviors of components designed to protect a program and its resources from unauthorized access, misuse, or attack. Common security features include access control, input validation, encryption, sandboxing, logging, and rate limiting. For instance, Nginx requires users to provide valid credentials (*e.g.*, username and password) before accessing restricted web pages. These features are particularly attractive targets for SUB attacks as compromising them often results in immediate, high-impact violations directly aligned with attacker objectives. In contrast, exploiting low-level primitives may produce large volumes of reports that are difficult to associate with meaningful, high-level security breaches. Figure 2(b) demonstrates the relationship among various attack goals. Prior control-flow attacks mainly pursue code execution (CE) or information leakage (IL), whereas our work focuses on constructing and leveraging security-oriented oracles (O_s) to evaluate semantic consequences of function substitutions.

We define a behavioral oracle as a pair (in, out), where in represents one or more test cases that trigger a specific security feature, and out denotes the observable program behavior reflecting the successful enforcement of that feature. Formally, $out := O_s(in)$, where O_s is the oracle function for security feature s. The in component also include necessary configurations to activate the security feature, such as compilation flags, runtime settings, command-line options, or environment-specific setup steps, like creating user credentials. For example, to test SQLite3 safe mode, we must launch the process with `-safe` and provide an input with a prohibited operation like `load_extension`. The output will contain an error like "cannot use the load_extension() function in safe mode".

To scale this process and minimize manual effort, we leverage large language models (LLMs) to assist in constructing behavioral oracles. As security has become a growing concern for users [32], developers frequently highlight newly implemented protections in publicly accessible materials, such as official websites [78], [42], [3], release notes, and source repositories. These materials provide valuable context that LLMs can use to synthesize expected behaviors and assist in oracle generation. We detail our LLM-assisted oracle construction in §V-A and evaluate its effectiveness in §VI-B.

C. Automatic Substitution and Measurement

We design an automated algorithm to systematically construct SUB attacks. This process relies on three key components: a target set from which substitution candidates are selected, a security oracle that verifies whether the security feature is properly enforced, and a substitution engine that systematically replaces allowed targets and checks for behavior deviations using the oracle. We obtain substitution candidates from the sub-ground truth set T^- described in §IV-A, and construct security-oriented oracles as discussed in §IV-B. Figure 2(c) provides an overview of the algorithm. For each indirect call $inst$ in a given application P, our framework substitutes the original runtime target with a candidate from T^-. The application is then executed with the test input in and the oracle O_s for security feature s. If the observed output deviates from the expected output defined by the oracle, we classify the substitution as a successful SUB attack. While our focus is on indirect function calls, the approach can be easily extended to other ICT instructions, including indirect jumps and returns.

In addition to validating the high-level security impact, we also consider the possibility that a substitution may corrupt the program's execution state, causing subsequent ICT instructions to target invalid functions, *i.e.*, functions outside of the valid target set T. Since such targets would violate precise CFI policies, the corresponding substitutions are not considered successful SUB attacks. To detect these cases, we monitor each ICT instruction during execution and verify whether its target remains within T^-. If any subsequent ICT targets fall outside of this set, we conservatively mark the substitution as a failure. This design avoids reporting attacks that would be blocked by precise CFI enforcement. While this conservative approach may discard feasible attacks due to incomplete coverage in T^-, it is acceptable to use as long as we can still construct a large number of SUB attacks across real-world programs.

V. SACK DESIGN & IMPLEMENTATION

We design and implement SACK, the first framework for systematically constructing SUB attacks at scale. Figure 3 shows the overall workflow. Given a target application, SACK first builds security-oriented oracles with the assistance of a large language model (§V-A). Each oracle defines the necessary configuration to enable a security feature, the input needed to trigger it, and the expected security behavior. SACK then executes the application with sample inputs to dynamically collect ICT targets triggered during benign execution (§V-B). These targets form the basis for candidate substitutions. The attack engine proceeds by selecting one oracle input and repeatedly executing the application (§V-C). In each run, it substitutes the target of a single indirect call with an alternative from the collected target set. After each substitution, SACK monitors the application's behavior and compares it against the expected output defined by the oracle. A deviation indicates that the substitution has compromised the security feature.

A. LLM-assisted Oracle Constructor

Creating a security-oriented oracle involves identifying relevant security features, synthesizing inputs (in) that enable and trigger those features, and defining monitors to capture the expected behaviors (out) that confirm correct enforcement. To automate this process, we employ a structured methodology that combines authoritative documentation sources with the reasoning capabilities of large language models (LLMs). Inspired by workflows from prior studies [29], [69], [53], [87], [94], [17], SACK constructs oracles via three steps: feature identification, document preparation, and oracle generation.

1) Feature Identification: In the first step, we identify security features supported by the target program using LLM assistance. To enhance the effectiveness of the model, we adopt a role-playing prompt engineering technique [48], instructing the LLM to act as a domain expert familiar with the program's design. The task is defined as listing all available security features. We employ a few-shot prompting strategy by providing example keywords such as "access control", "input validation", and "logging", to guide the model's response. Upon receiving the prompt, the LLM generates a list of security features based

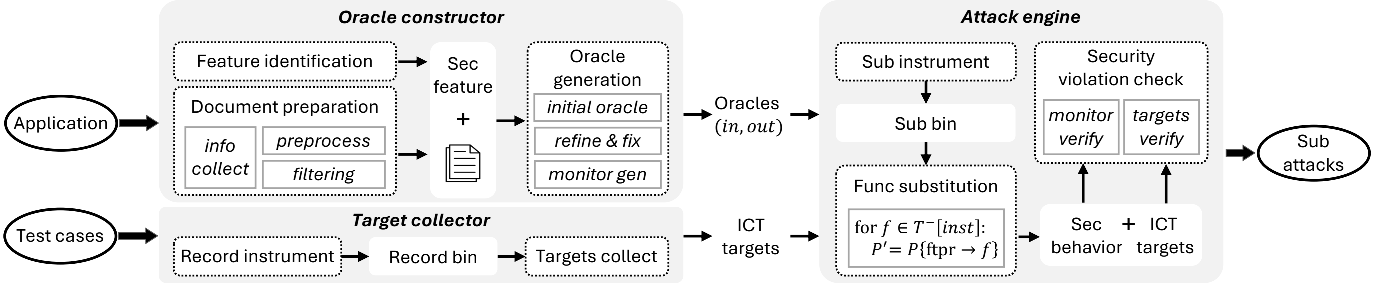


Fig. 3: Sack framework. Given a program, it first builds security-oriented oracles with LLM assistance and collects runtime-triggered targets of ICT instructions; then, Sack uses oracles and ICT targets to automatically construct Sub attacks.

on its embedded knowledge and, if available, online sources. For example, when querying about SQLite3, the LLM identifies security features such as database encryption, SQL-injection protection and read-only database support. One particularly relevant result is "safe mode (.safe command)", described as "safe mode restricts the CLI from running potentially dangerous commands and loading extensions", which corresponds to the feature highlighted in our motivating example (Figure 1). We collect all returned features for use in the subsequent steps.

2) Document Preparation: In parallel with feature identification, we collect detailed documentation from the application's official website to support concrete oracle construction. Relying solely on LLM-generated responses is often insufficient, as oracles usually require specific inputs and configuration settings to reliably trigger the targeted security feature. Although LLMs are trained on extensive data, they may lack detailed, implementation-specific knowledge such as compilation flags, runtime configurations, or command-line arguments. In such cases, official documentation provides authoritative, context-specific information to fill these gaps. Since our oracles aim to capture high-level security behaviors, we focus on manual-style documents that guide users on configuring and invoking security features. These materials typically contain concrete commands, configuration options, and usage examples. For example, the SQLite3 command-line shell documentation at https://sqlite.org/cli.html explains that the `-safe` flag enables safe mode and lists prohibited actions, like `load_extension` and `readfile`. This information helps LLM understand both how to enable the feature and how to verify its enforcement.

To automate document collection, we develop a crawler to recursively download all reachable pages from a user-provided entry point, treating each page as a standalone document. Since crawled pages may contain irrelevant content, we apply a two-step preprocessing and filtering process to extract useful information. First, we remove non-content elements such as navigation bars, layout structures, and scripts. Then, we perform keyword-based filtering to retain only security-related documents. We search for terms such as "authentication", "protection", "security", "logging", "password", and their synonyms. Documents matching any of these terms are retained for subsequent oracle construction, while others are discarded.

3) Oracle Generation: We use an LLM to parse security-related documents and generate behavioral oracles for identified security features. This process contains three steps: initial oracle generation, refinement and correction, and monitor definition.

Initial oracle generation. We prompt an LLM to generate initial security oracles by providing a structured query that contains a system prompt, a user prompt, a target security feature, and all collected documentation. The system prompt adopts a role-playing strategy, instructing the LLM to act as an experienced software developer. The task is to generate a security oracle containing six components, when applicable.

- Compilation flags required during compilation to enable the security feature. For example, enabling SSL/TLS in Nginx requires the `--with-http_ssl_module` flag.
- Configuration directives like runtime settings or initialization parameters to activate the feature. For instance, basic authentication in Nginx requires setting the `auth_basic` and `auth_basic_user_file` directives in the configuration file.
- Additional commands for enabling the feature. For example, basic authentication in Nginx requires a credential file, which can be generated using a tool such as `htpasswd`.
- Legal input and expected outcome under the feature. For example, SQLite3 in safe mode should execute valid SQL queries without prohibited actions successfully.
- Illegal input and expected outcome under the feature. For instance, SQLite3 in safe mode should reject invalid SQL queries with prohibited actions and print an error message.
- Final judgment for verifying the feature enforcement. For SQLite3, we will scan the program output for errors to determine whether the safe mode is properly enforced.

Each LLM query pairs one identified security feature with all relevant documents to enable contextual reasoning.

Refinement and correction. After obtaining initial oracles, we test them by executing the outlined steps. If any discrepancies or errors occur, we collect the observed behaviors and submit them along with the original oracle to the LLM for refinement. This iterative process is repeated up to three times. If errors persist, we manually intervene to correct the oracle. We evaluate the effectiveness of this process and report the result in §VI-B.

Monitor development. Once validated, we develop monitors using Python to automatically detect whether the oracle's expected behavior (*out*) occurs. We categorize monitoring techniques into two types based on where to find the observable behavior. The *log-based* approach captures behaviors recorded in the program's internal logs. For example, in Nginx and Apache, HTTP status codes (*e.g.*, `200` for success) are saved

```
1   ; instrument indirect call for collecting targets
2 + call @do_record(%63, 142)       ; 142 is the icall ID
3   %72 = call %63(%65, %70)
4
5   ; substitute one function pointer (sub attack)
6   %59 = getelementptr %58, 0, 0
7 + call @do_sub(%59, 142)          ; 142 is the icall ID
8   %63 = load %59
9   %72 = call %63(%65, %70)
```

Fig. 4: Instrumentation to LLVM IR for recording and substitution

```
1   void do_record(long addr, long icall_id) {
2     FILE * fd = fopen("...", "w");
3     long pair[2] = {icall_id, addr};
4     fwrite(pair, sizeof(pair), 1, fd);
5     fclose(fd);
6   }
7   void do_sub(long * mem_addr, long icall_id) {
8     if (icall_id == atol(getenv("ID_TO_SUB")))
9       *mem_addr = atol(getenv("ADDRESS_TO_SUB"));
10  }
```

Fig. 5: Extra runtime code added for recording and substitution

in access logs, reflecting the enforcement of features such as authentication. To monitor such information, we analyze the corresponding log files. The *output-based* approach captures behaviors printed directly to standard output `stdout` or standard error `stderr`. In this case, we redirect the program's output to a dedicated file for examination. We choose the monitoring method based on the form and location of the observable output. For example, attempting to use `load_extension` in SQLite3 safe mode produces an error message, which we detect by redirecting output to a file and analyzing its contents.

B. ICT Targets Collector

To collect allowed ICT targets triggered during benign executions, SACK uses the LLVM compiler `clang` to instrument the program and produces a binary for target recording. Specifically, SACK assigns each indirect call instruction (*icall*) a unique identifier, referred to as the *icall ID*. Before each icall, SACK inserts new LLVM intermediate representation (IR) instructions to invoke a custom recording function, which takes two arguments: the function pointer (*i.e.*, the current target) and the corresponding icall ID. During execution, this recording function logs each observed (icall ID, target) pair to a trace file. After execution completes, SACK merges all trace files into a global map, where each icall ID is associated with the set of target functions it invokes during benign runs.

Figure 4 and Figure 5 illustrate the IR-level instrumentation and the supporting helper functions added to the program. For readability, we omit verbose type information in the LLVM IR. In Figure 4, line 3 shows the LLVM IR corresponding to the indirect call at line 4 in Figure 1, where `%63` holds the function pointer `do->xAuth`. Line 2 demonstrates the instrumentation inserted before the call, *i.e.*, a call to `do_record` with `%63` and a constant icall ID `142` as arguments. Figure 5 (lines 1-6) presents a simplified version of the `do_record` implementation, which logs the icall ID and target address to a trace file. We will compile this function into the final program binary to support target recording. The entire instrumentation process is fully automated and does not require manual intervention.

Recording all icalls and their targets during execution can generate large trace files and introduce significant disk I/O overhead. We observe, however, that many icalls repeatedly invoke the same target along the execution. To reduce redundancy and improve performance, we cache the most recently recorded target for each icall in an in-memory array. For each icall, a new log entry is written only when the target differs from the last recorded one. This optimization significantly reduces unnecessary logging and improves runtime efficiency.

C. Automatic Attack Engine

With synthesized security oracles and recorded ICT targets, SACK constructs and evaluates SUB attacks by modifying function pointers before indirect calls. It enumerates valid substitutions and assesses whether any weakens or disables a security feature. For each attack, SACK also verifies that all subsequent indirect calls conform to a precise static CFI policy.

1) Function Substitution: For each security oracle, SACK executes the program multiple times with the same input in, once for each recorded alternative target of every indirect call (icall). In each execution, SACK selects a single icall and replaces its original target with a different one observed during benign runs. We observe that modifying multiple function pointers within the same execution often leads to instability, like crashes or hangs. To ensure reliable testing, the current design limits each execution to a single substitution.

To simulate SUB attacks, SACK uses an additional LLVM pass during compilation to enable runtime modification of function pointers. This pass performs two tasks. First, it identifies each icall instruction and conducts backward data-flow analysis to find the memory address of the function pointer being invoked. Second, it instruments the code just before the pointer is loaded by inserting a call to a helper function `do_sub`, passing the pointer's memory address and the icall ID as arguments. Lines 6-9 in Figure 4 show an example of this instrumentation for the indirect call at line 4 in Figure 1. Line 6 computes the address of `db->xAuth` and stores it in `%59`; line 8 loads the pointer into `%63`; and line 9 performs the indirect call. The call to `do_sub`, inserted before line 8, allows runtime substitution of the function pointer using environment variables. Lines 7-10 in Figure 5 present a simplified implementation of `do_sub`, which checks the icall ID and modifies the memory accordingly. Before each execution, SACK updates environment variables to specify the target substitution. By iterating over all combinations of icall IDs and valid targets, SACK systematically enumerates all possible substitutions across all security oracles.

Limiting Repetitive Substitutions. While our brute-force substitution strategy is conceptually simple and domain-agnostic, repeated substitutions at highly frequent indirect-call sites may lead to excessive program re-executions. For an icall with M valid targets that is dynamically invoked N times, the original strategy performs $M \times N$ substitutions, which can be prohibitively large when N is high. To mitigate this cost, we introduce a lightweight heuristic that bounds the number of dynamic invocations considered for substitution at each call site. Instead of substituting across all N invocations, we examine only the first $min(N, \tau)$ invocations, where τ is a configurable threshold. This design focuses explorations on early executions, which often reflect the most representative execution contexts while avoiding redundant work caused by hot call sites inside loops or event-driven paths. With the threshold, the total substitution per icall become $M \times \min\{N, \tau\}$, significantly reducing the exploration cost. This optimization improves efficiency at the potential expense of pruning some feasible SUB attacks. We quantify both the efficiency gains and the corresponding impact on attack coverage in (§VI-C2).

Support for C++ virtual calls. SACK operates at the LLVM IR level, making it applicable to both C and C++ programs. However, C++ virtual calls require special handling due to the use of virtual function tables (vtables). In C, each indirect call usually uses a standalone function pointer, so modifying it only affects one target. In contrast, C++ vtables group function addresses into read-only memory regions, and the program accesses virtual methods via a vtable pointer and an index. This setup makes runtime modification challenging: individual function addresses in the vtable are read-only, and changing the vtable pointer affects all methods in the class. In real-world C++ exploits, attackers often craft a fake vtable and overwrite the vtable pointer. To simplify our framework, SACK modifies the program binary to make all loadable segments writable using the LIEF library [54]. This allows SACK to simulate realistic attacks by directly modifying vtable entries as needed, using the same substitution technique shown in Figure 4.

2) Security Violation Check: SACK performs two levels of checks to determine whether a SUB attack has successfully compromised a security feature. First, SACK invokes the corresponding oracle monitor to assess whether the feature remains correctly enforced. It analyzes program's logs or standard output/error to compare observed behavior with the oracle's expected outcome. Any deviation from the expected output indicates a violation, suggesting that the substitution has disrupted or disabled the security feature. Second, SACK ensures that all subsequent icalls comply with a precise CFI policy. While the initial substitution is chosen from the valid target set to avoid CFI violations, it may place the program into an inconsistent or unintended state. This state may cause subsequent icalls to target functions not allowed under precise CFI, thereby violating CFI guarantees. To detect this, SACK monitors all downstream ICT instructions and checks whether their targets fall within the previously collected valid set. If no, the execution is flagged as a failure, and the corresponding substitution is not counted as a successful attack.

VI. EVALUATION

We evaluate the SACK framework on real-world applications to assess the prevalence of SUB attacks, and to measure the effectiveness of our systematic construction approach. Our evaluation aims to answer the following research questions.

- Can SACK effectively construct SUB attacks? (§VI-A)
- Can the oracle generator produce accurate oracles? (§VI-B)
- Can the attack engine reliably substitute targets? (§VI-C)
- Can SACK support statically inferred CFI targets? (§VI-D)

Programs for evaluation. Since no standard benchmark exists for evaluating CFI attacks, we follow the methodology of prior work [11], [26], [82] to select a representative set of seven widely used C/C++ applications spanning diverse domains. Their source-code sizes range from 132K to 4.7M lines of code (LoC). Specifically, Nginx and Apache are widely deployed web servers for handling HTTP requests; SQLite3 is a lightweight and extensively used database engine; ProFTPD is a secure FTP server; Sudo is a critical security utility for privilege escalation management in Unix-like systems; Wireshark is a popular network protocol analyzer for traffic capture and inspection; and V8 is a high-performance JavaScript (JS) engine, used in browsers and server-side environments. All applications incorporate various security features to ensure safe and robust execution. Table I summarizes their key characteristics.

Experiment environment. We conduct all experiments within Docker containers running 64-bit Ubuntu 20.04. Each container uses default settings and has full access to the host machine's CPU and memory resources. The host machine runs 64-bit Ubuntu 22.04 with a 56-core Intel(R) Xeon(R) Gold CPU, 500 GB RAM, and 1 TB SSD. For LLM-assisted oracle construction, we utilize GPT-4.1 released on 2025-04-14. Unless otherwise noted, we set the substitution threshold τ to 1,000 to balance efficiency with comprehensive attack coverage.

Inputs for ICT target collection. To collect indirect call targets, we adopt a lightweight strategy to identify test cases. When official test suites are available, we utilize them directly. Otherwise, we prompt the LLM to synthesize one or two minimal test cases that exercise basic program functionality. In addition, oracle inputs from §V-A are reused to ensure coverage of security-relevant code paths. We deliberately adopt this minimal-effort approach to reflect a realistic attacker setting, where the goal is to assess whether SUB attacks can be constructed using publicly available or easily generated inputs, without requiring exhaustive testing infrastructure.

A. Constructed SUB *Attacks*

SACK successfully constructs SUB attacks across all seven evaluated programs, using a total of 22 synthesized security oracles. To quantify these attacks, we adopt three counting strategies: *(i)* counting each unique indirect call (icall) as a distinct attack, regardless of the original or substituted targets; *(io)* counting each unique pair of (icall, original target), ignoring the substituted target; *(ion)* treating each unique tuple (icall, original target, new target) as a distinct attack. Using these respective methods, SACK constructs 52, 75, and 419 attacks. In the remainder of the evaluation, we report attack counts using the most precise method (ion), as it best reflects the diversity of successful substitutions. Compared to the two SUB attacks in the CFB work [11], our results reveal the prevalence of SUB attacks in real-world applications and the feasibility of constructing them using a systematic and automated approach.

Table I summarizes the target programs, versions, code sizes, supported security features, synthesized oracles, and number of successful attacks. A dash (-) indicates that SACK did not generate any successful attack for the corresponding oracle.

TABLE I: Tested programs, security oracles and constructed Sub attacks. We adopt three counting strategies: (i) counts unique indirect calls; (io) counts unique pairs of indirect call and original target; (ion) counts unique triplets of indirect call, original target and substituted target. **-** means no attacks are constructed. * indicates we construct real-world attacks with concrete historical vulnerabilities. For kLoC, we use the cloc tool on the source code prior to compilation, summing the lines of C, C++ and C/C++ headers, where applicable.

Program	Version	kLoC	Lang.	Security feature	Oracle	#Sub:	(i)	(io)	(ion)
Nginx	1.27.1	165	C	Authentication	**(N1)** Accessing a restricted webpage is rejected	*	6	10	41
				Rate limiting	**(N2)** Send two continuous requests; only one is accepted	*	9	11	82
				Web app firewall	**(N3)** Sending an SQL-injection request is rejected		6	6	25
				Restrict methods	**(N4)** Accessing a webpage w/ a blocked method is rejected	*	6	15	50
				Logging	**(N5)** Accessing a webpage is recorded		8	8	52
				SSL/TLS	**(N6)** HTTPS packets are encrypted		-	-	-
SQLite3	3.47.2	290	C	Unsafe commands	**(Q1)** Executing an unsafe SQL command is blocked	*	3	3	33
				Read-only mode	**(Q2)** Modifying a read-only schema is rejected		-	-	-
ProFTPD	v1.3.8c	242	C	Authentication	**(P1)** Login w/o password is rejected		1	2	20
				Login attempt limit	**(P2)** Exceeding the password attempt limit is banned		2	4	47
				User permission control	**(P3)** Executing forbidden commands is denied		1	1	7
				Auth-required actions	**(P4)** Executing sensitive actions w/o login is denied		-	-	-
Sudo	1.9.16	132	C	Logging	**(U1)** Using sudo is recorded		1	1	2
				Extra approval	**(U2)** Using sudo w/o satisfying extra approval is rejected		1	1	1
				Authentication	**(U3)** Using sudo w/o password is rejected		-	-	-
Apache	2.4.63	213	C	Authentication	**(A1)** Accessing a restricted webpage is rejected		2	2	22
				Web app firewall	**(A2)** Sending a SQL-injection request is rejected		1	1	3
				Restrict methods	**(A3)** Accessing a webpage w/ a blocked method is rejected		1	1	4
				Logging	**(A4)** Accessing a webpage is recorded		1	1	6
				Block malicious URL	**(A5)** Sending a request w/ malicious URL is blocked		1	2	11
Wireshark	4.4.5	4,713	C/C++	Malform detection	**(W1)** Malformed packets are highlighted in expert information		1	5	12
V8	10.7.190	1,697	C++	Block unsafe method	**(V1)** Executing `Shell::System` is rejected	*	1	1	1
Total		7,450			22		52	75	419

We also construct five end-to-end Sub attacks using historical bugs in SQLite3 against oracle Q1 (based on CVE-2017-6983), V8 against oracle V1 (based on CVE-2021-30632), and Nginx against oracle N1/N2/N4 (based on CVE-2013-2028) marked with an asterisk (*), with details of three provided in the following case studies. As these vulnerabilities have been patched for years, our demonstrations pose no immediate threat.

Across the 22 oracles, Sack successfully constructs Sub attacks that bypass 18, demonstrating a broad range of security consequences. Authentication-related oracles (Nginx-N1, ProFTPD-P1/P3/P4, Sudo-U3, Apache-A1) restrict access to privileged resources or operations. Sack constructs attacks compromising four of these six features. Connection-limiting oracles (Nginx-N2, ProFTPD-P2) enforce rate limits on requests or logins. Sack builds attacks that bypass both protections, enabling unrestricted brute-force attempts. Logging-related oracles (Nginx-N5, Sudo-U1, Apache-A4) record security-relevant events. Sack builds Sub attacks against all three, allowing adversaries to erase traces of malicious activity. Dangerous operation oracles (Nginx-N3/N4, SQLite-Q1/Q2, Sudo-U2, Apache-A2/A3/A5, V8-V1) aim to block high-risk operations by default. Sack constructs attacks that bypass eight of the nine protections in this category. Program-specific oracles include Nginx-N6, which enforces SSL/TLS encryption, and Wireshark-W1, which highlights malformed packets. Sack constructs a successful Sub attack against Wireshark-W1.

While Sack constructs 19.0 attacks per oracle on average, Table I reveals significant variation across oracles. In extreme cases, such as Nginx-N2, Sack identifies 82 distinct substitutions to bypass the rate-limiting feature, while for Nginx-N6, SQLite-Q2, ProFTPD-P3, and Sudo-U3, no attacks are found. We attribute this variation to several factors, like application size, feature complexity, and programming style (*e.g.*, the extent to which function pointers are used). Interestingly, attack counts are more consistent across oracles within the same program, suggesting that program-level characteristics have a stronger influence on attack feasibility than individual feature properties.

Next, we present three case studies that demonstrate the severity and prevalence of Sub attacks, and the effectiveness of our Sack platform. For each of these cases, we construct working Sub exploits with concrete real-world vulnerabilities.

Case study 1: disarming SQLite3 safe mode. As discussed in §II-A, SQLite3 provides a safe mode to restrict the execution of untrusted actions, such as loading external extensions. Using Sack, we automatically construct 33 Sub attacks that effectively disable safe mode and re-enable previously prohibited actions. These attacks involve three distinct icall instructions. One such attack corresponds to the example attack presented in §II-C, where the function pointer `do->xAuth` at line 4 of Figure 1 is substituted from `safeModeAuth` to a valid alternative, `idxAuthCallback`. The remaining two icalls appear earlier along the call path to `sqlite3AuthCheck`, which indirectly invokes `safeModeAuth`. By substituting these functions with alternative valid targets, attackers bypass the execution of `sqlite3AuthCheck` entirely and disable safe mode.

We develop a full exploit based on CVE-2017-6983, a historical type-confusion vulnerability in SQLite3 that enables arbitrary memory writes [28]. Since the vulnerability was patched in 2017, we manually revert the fix to reintroduce the bug. The exploit proceeds in three steps. First, it leverages the type confusion to craft an arbitrary-write primitive. Second, it uses this primitive to overwrite `do->xAuth`, replacing `safeModeAuth` with `idxAuthCallback` and thereby disabling safe mode. Finally, it invokes `load_extension` via a `SELECT` clause, causing SQLite3 to load a malicious shared library. This library contains a payload in its ELF constructor, which automatically executes upon loading. With this attack, we achieve arbitrary code execution in SQLite3 safe mode.

Case study 2: escaping V8 sandboxing. The V8 JavaScript shell, d8, enforces a sandboxing mechanism that disables access to the operation `os.system` which allows the execution of arbitrary system commands. To invoke `os.system`, users must explicitly launch d8 with the `--enable-os-system` flag. However, certain operations, like `os.rmdir` which deletes directories, remain accessible without this flag. SACK constructs a SUB attack that exploits this gap by substituting the function pointer for `os.rmdir` originally pointing to `v8::Shell::RemoveDirectory`, with `v8::Shell::System`, the handler for `os.system`. As a result, even when d8 is launched without the special flag, invoking `os.rmdir` silently executes arbitrary commands via `os.system`, bypassing the restriction. Importantly, this attack does not violate even a perfectly precise CFI policy as both functions are valid targets of the indirect function call responsible for dispatching `os.*` operations.

We construct an end-to-end exploit using CVE-2021-30632, a logic bug in V8's just-in-time (JIT) compiler that leads to type confusion and arbitrary memory modification. This vulnerability was patched in 2021, and we revert V8 to version 9.3.345.16 (released 2021-09-01) to reintroduce the flaw. We build upon a publicly available proof-of-concept to reproduce the arbitrary write primitive. To integrate the SUB attack, we target the `os.rmdir` object, which is implemented as a `JSFunction` in the V8 source. We traverse its internal structure chain: `JSFunction->SharedFunctionInfo->FunctionTemplateInfo->CallHandlerInfo->Foreign` to locate the underlying function pointer. Notably, the offset between `RemoveDirectory` and `System` is fixed, even with ASLR enabled. This allows us to redirect the pointer by simply adding a known constant offset to the existing address, without requiring knowledge of the actual address of `System`. Once the pointer is overwritten, the exploit simply invokes `os.rmdir("bash",[])`, which results in a shell being spawned.

Case study 3: bypassing Nginx authentication. SACK identifies 41 distinct substitutions that bypass Nginx HTTP basic authentication. In particular, Nginx-N1 requires a valid username and password to protect sensitive web pages, while our SUB attacks enable unauthorized access without knowledge of valid credentials. We analyze and categorize these attacks into three groups based on attack internals. First, five attacks cause Nginx to skip the authentication phase entirely, allowing unrestricted access. We construct a full exploit using CVE-2013-2028, by substituting `ngx_http_core_find_config_phase` with `ngx_http_core_post_rewrite_phase`. Second, one attack replaces the password-based auth function with a weaker IP-based alternative, thereby reducing the security quality. Third, the remaining 35 attacks target functions responsible for parsing Nginx configuration files. By replacing the parser with unrelated functions, these attacks prevent the correct loading of authentication settings, effectively disabling access control.

B. Performance of Oracle Generation

We evaluate the performance of our LLM-assisted oracle generator discussed in §V-A, and present results in Table II.

Using few-shot prompting, our feature-identification step yields 10 to 20 security features for each program. Due to resource constraints, we select a representative subset of these features for evaluation, focusing on commonly used protections, like authentication, authorization, logging, and sandboxing.

We report three metrics during document preparation: the crawler entry point (entry), the number of pages collected (#crawl), and the number retained after preprocessing (#filter). From the entry point, the crawler collects eight to 142 pages, depending on two factors. First, the choice of entry point significantly affects the collected pages. Instead of choosing the root directory on the official website, we prioritize manual-like documentation that describes how users can configure and invoke security features. For example, the Wireshark user guide links to 196 pages, most of which pertain to the GUI version. By instead using the manual page of Tshark, the terminal-based version, we get only eight relevant documents. Second, document structure and application design also influence the collected page. For example, Apache offers 126 module descriptions, each presented as a separate page. Users can customize the entry point to include more or fewer documents, depending on how extensively they wish to construct SUB attacks. The filtering process eliminates 19.2% to 63.0% of collected documents. For SQLite3, up to 63.0% of pages are excluded due to missing security keywords. These pages typically cover general functionality or serve as introductory content. In contrast, only 19.2% of ProFPTD's pages are filtered out, because terms like "logging" and related keywords appear frequently. This filtering ensures our tool focuses on security-relevant content and reduces noise in subsequent LLM queries.

Among 22 features, SACK successfully generates 20 oracles without requiring extra documents. Only two features, Nginx-N3 and Apache-A2, initially fail due to missing information about the third-party web application firewall (WAF) module. Providing these documents yields successful oracle generation.

We assess the quality of initial oracles across six components: compilation flags (compile), configuration directives (config), additional commands (cmd), legal and illegal input and outcomes, and final judgment (final). Each component is evaluated using a three-level scale: ✓ if we can apply it as-is; "auto-fix" if it contains errors but can be resolved by three LLM re-queries; "manual" if it cannot be resolved within three LLM re-queries and requires human investigation.

Out of 22 oracles, 18 have all components marked as ✓, indicating the initial oracles successfully guide the setup and verification of the security features. Three oracles, Apache-A1/A2/A5, contain components marked as "auto-fix". We identify that the primary causes for these issues are idealized assumptions and omissions of critical steps. Although the LLM is instructed to provide step-by-step commands, it occasionally assumes that certain requirements are already satisfied, which may not hold in practice. For Apache-A2, enabling its WAF

TABLE II: Statistics of LLM-assisted oracle generation. We report the crawler entry, number of collected pages (#crawl), retained after filtering (#filter), and extra entries. We measure oracle quality across compilation flags, configuration directives, extra commands, legal/illegal input/output, and final judgment. We rate them as: usable as-is (✓), auto-fixable, or manually fixable. We record the time cost for document prep, feature identification, oracle generation (init), auto-fix, and manual correction. We report the token length and the average oracle cost.

Program	**Documentation**			**Oracle**	**Initial Oracle Quality**							**Prep Time**	**LLM Time**			**Man Time**	**LLM Query**	
	entry	#crawl	#filter		extra	compile	config	cmd	legal	illegal	final		feat	init	fix		#token	price
Nginx	↗	67	29									36″	16″				138,973	0.278
				N1	-	✓	✓	✓	✓	✓	✓			34″	-	-		
				N2	-	✓	✓	✓	✓	manual	✓			19″	-	4′		
				N3	↗	✓	✓	✓	✓	✓	✓			41″	-	-		
				N4	-	✓	✓	✓	✓	✓	✓			12″	-	-		
				N5	-	✓	✓	✓	✓	✓	✓			20″	-	-		
				N6	-	✓	✓	✓	✓	✓	✓			15″	-	-		
SQLite3	↗	46	17									12″	17″				129,303	0.259
				Q1	-	✓	✓	✓	✓	✓	✓			10″	-	-		
				Q2	-	✓	✓	✓	✓	✓	✓			9″	-	-		
ProFTPD	↗	99	80									39″	11″				243,125	0.486
				P1	-	✓	✓	✓	✓	✓	✓			38″	-	-		
				P2	-	✓	✓	✓	✓	✓	✓			19″	-	-		
				P3	-	✓	✓	✓	✓	✓	✓			22″	-	-		
				P4	-	✓	✓	✓	✓	✓	✓			22″	-	-		
Sudo	↗	49	36									25″	23″				193,509	0.387
				U1	-	✓	✓	✓	✓	✓	✓			13″	-	-		
				U2	-	✓	✓	✓	✓	✓	✓			15″	-	-		
				U3	-	✓	✓	✓	✓	✓	✓			15″	-	-		
Apache	↗	142	86									43″	13″				289,812	0.580
				A1	-	✓	✓	✓	auto-fix	✓	✓			31″	5″	-		
				A2	↗	auto-fix	✓	auto-fix	✓	✓	✓			21″	26″	-		
				A3	-	✓	✓	✓	✓	✓	✓			16″	-	-		
				A4	-	✓	✓	✓	✓	✓	✓			29″	-	-		
				A5	-	✓	manual	✓	auto-fix	auto-fix	✓			19″	6″	7′		
Wireshark	↗	8	6	W1	-	✓	✓	✓	✓	✓	✓	3″	10″	15″	-	-	31,280	0.063
V8	↗	61	27	V1	-	✓	✓	✓	✓	✓	✓	11″	5″	17″	-	-	48,726	0.097

feature requires two third-party modules, libmodsecurity and Modsecurity-apache Connector. However, the LLM assumes their availability and omits necessary compilation flags to enable them. Another issue in Apache-A2 is that the initial *cmd* component fails to include the critical step of setting *SecRuleEngine On*, which results in alerts being logged but packets not blocked. Apache-A5 blocks malicious packets based on packet content, but the *legal* and *illegal* components initially provide only binary packet representations, without explaining how to send them to Apache. These incorrect assumptions and omissions are recoverable by at most three LLM re-queries.

Two oracles, Nginx-N2 and Apache-A5, require manual intervention to produce functional results. In both cases, the LLM fails to identify the root causes and resolve them within three re-queries. Nginx-N2 enforces rate limiting, which restricts how frequently a client can send requests within a defined time window. The oracle's *illegal* component sends five requests in a loop, expecting the first two to be accepted and the remaining three to be blocked. This aligns with the *config* component, which sets a limit of one request per second with a burst of two. However, due to network latency and timing variability, this actual behavior is inconsistent, which occasionally accepts three requests instead of two. To address this issue, we manually reduce the request count to two, and update the configuration to allow only one request per minute with no burst. This produces a stable outcome where the first request is accepted and the second is consistently rejected. For Apache-A5, the feature relies on a regular expression in the configuration file to block malicious requests. However, the initial *config* component includes a flawed regular expression that matches all inputs, causing the system to block all requests. We manually correct the issue to produce a functional oracle.

We record the document preparation time (Prep Time), as well as the time spent on LLM API queries during feature identification (feat), initial oracle generation (init), and auto-fix attempts (fix). For oracles that cannot be automatically fixed, we also report the manual correction time (Man Time). Document preparation, which includes crawling, preprocessing, and filtering, completes within 45 seconds (″) for all programs. The preparation time mainly depends on the number and length of the collected pages. Feature identification completes within 25″ for all cases. Initial oracle generation typically takes less than 30″ per oracle, with a few cases requiring between 30″ and 40″. Auto-fix attempts are similarly efficient, taking under 30″ per oracle. When manual intervention is necessary, the correction time remains modest: fixing Nginx-N2 takes approximately 4 minutes (′) and Apache-A5 requires around 7′. These results demonstrate the overall efficiency and practicality of our LLM-assisted oracle generation process.

Since document length dominates the total query size, we compute the token length of the filtered documentation (#token) and use it to estimate the generation cost for each oracle (price). In our evaluation, the cost of generating each oracle remains below $0.60. This cost can be further reduced by trimming the input documents or using a lower-cost LLM.

TABLE III: Statistics of attack generation. We count all/triggered/substituted icalls, collected targets, SUB attempts, attacks breaking oracles (A_0), without CFI violations (A_1), merging invocations (A_2) and manually confirmed (A_3). We also record the times used for all SUB attempts (T), each attempt (T/s), and one successful attack (T/A_3). T^+_{TFA} and T^+_{MLTA} denote statically inferred targets from TFA and MLTA.

O_s	**iCalls & Targets**				**SUB Attacks**					**Time Cost**			T^+_{TFA}		T^+_{MLTA}	
	all	triggered	substituted	target	attempt	A_0	A_1	A_2	A_3	T	T/s	T/A_3	A_3	time	A_3	time
N1	339	57	28	263	6,208	41	41	41	41	476″	0.077″	11.6″	61	796″	67	829″
N2	339	69	30	270	10,194	145	111	90	82	1450″	0.142″	17.7″	180	2550″	189	2803″
N3	351	65	33	291	10,473	62	25	25	25	740″	0.071″	29.6″	82	1183″	90	1232″
N4	339	60	29	287	7,294	55	55	50	50	555″	0.076″	11.1″	62	934″	66	938″
N5	351	67	35	296	10,728	74	69	69	52	824″	0.077″	15.8″	150	1156″	169	1223″
Q1	456	21	5	53	193	34	34	33	33	1″	0.005″	0.03″	33	5″	33	8″
P1	573	36	11	132	10,834	20	20	20	20	3837″	0.354″	191.9″	86	8991″	109	10568″
P2	573	37	11	137	12,445	59	47	47	47	17012″	1.367″	362.0″	205	30528″	243	36686″
P3	573	50	12	136	16,971	7	7	7	7	6815″	0.402″	973.6″	20	8476″	23	12250″
U1	368	67	8	25	146	4	2	2	2	6″	0.041″	3.0″	31	48″	31	69″
U2	380	58	7	23	124	1	1	1	1	6″	0.048″	6.0″	4	28″	8	68″
A1	1,124	186	49	414	27,359	86	43	22	22	6006″	0.220″	273.0″	146	8033″	215	11024″
A2	1,124	189	51	343	15,478	8	3	3	3	11543″	0.746″	3847.7″	37	13914″	164	21619″
A3	1,124	179	46	409	27,316	10	4	4	4	6434″	0.236″	1608.5″	94	7256″	185	10235″
A4	1,124	186	49	415	27,435	38	11	11	6	6501″	0.237″	1083.5″	10	6791″	118	10091″
A5	1,124	176	44	393	27,265	26	11	11	11	5858″	0.215″	532.5″	15	8495″	77	10975″
W1	868	74	36	197	50,562	42	12	12	12	30239″	0.598″	2519.9″	165	122162″	176	231877″
V1	5,998	285	92	540	6,116	1	1	1	1	839″	0.137″	839.0″	1	12148″	1	16696″

C. Performance of Attack Engine

To evaluate the performance of the attack engine, we collect various statistics during ICT target collection and automatic substitution, as summarized in Table III. Each row corresponds to one oracle for which SACK successfully constructs SUB attacks. For each oracle, we count the total number of statically available icalls, dynamically triggered icalls, and those with multiple targets observed at runtime (thus used in substitution attempts). The number of available icalls can vary across oracles due to differences in compilation flags, which may enable or disable certain features. Similarly, input diversity may exercise different modules, resulting in varying numbers of triggered icalls. We perform substitution only on icalls with two or more observed targets. We also record the number of target functions across all substituted icalls, as well as the number of substitution attempts. Finally, we track the number of attacks at four levels: violating the oracle-define behavior (A_0), conforming to precise CFI (A_1), unique icalls (A_2), and confirmed manually (A_3).

On average, the oracle inputs achieve 13.7% coverage of all icalls. For icalls with multiple targets, the coverage drops to 4.6%. The low coverage suggests that our evaluation may miss additional attacks, and thus underestimate the full prevalence of SUB attacks. Nevertheless, our primary goal is not exhaustive enumeration, but rather to demonstrate the feasibility and widespread applicability of SUB attacks. Despite the limited coverage, SACK successfully constructs 419 attacks, providing compelling evidence of both feasibility and prevalence. We further analyze the underestimation issue in §VII-D.

During each execution, a single icall may be invoked multiple times. SACK performs all valid substitutions per invocation, resulting in 267,141 pointer corruptions and program re-executions. Among them, 713 substitutions cause deviations from the oracle-defined behavior, like bypassing authentication. In 216 cases, the modified execution triggers previously unseen ICT targets; to ensure compatibility with a precise CFI policy, SACK conservatively treats these as attack failures. After merging repeated invocations of the same icall, we obtain 449 unique attacks, of which 419 are manually verified as genuine. The remaining 30 are identified as false positives primarily due to inaccuracies in our implementation of behavior monitors. We further analyze these monitor issues in §VII-E.

1) Time Cost of Attack Engine: Table III summarizes the time cost of automatic substitutions, including total time, average time per substitution (T/s), and per confirmed attack (T/A_3). The per-attack cost T/A_3 varies across programs and oracles. For SQLite3 and Sudo, all confirmed attacks are constructed in under 5″ on average. Nginx requires slightly more time, averaging 18.7″. The time increases to 8.4′ for ProFTPD. V8, Apache, and Wireshark require more time, *i.e.*, 14′, 19′, and 46′, respectively, due to increased complexity and substitution volume. Prior work like control-flow bending [11] and newton [82], does not report their time costs. Control Jujutsu [26] reports construction times of 1″, 10″, 14″ and 15′ for vsftpd, HTTPD, Nginx, and BIND, respectively. Therefore, SACK achieves comparable efficiency, demonstrating that its automated approach performs competitively with prior efforts.

We identify three key factors that influence the time cost of the attack engine: the nature of the security feature, program complexity, and the number of substitution attempts. First, some security features require extended execution to verify. For example, ProFTPD-P2 enforces login rate limiting by banning users after several failed attempts. Verifying this behavior requires multiple login failures, resulting in the highest per-attempt time (T/s) of 1.367″ among all evaluated cases. Second, program complexity impacts execution time. For instance, Wireshark takes approximately 0.6″ to parse a PCAP file containing just ten packets, significantly increasing the cost of each substitution attempt. Third, the number of attempts scales with the number of triggered icalls and available targets. Although Wireshark triggers a typical number of unique icalls, one specific icall with 80 targets is invoked 6,480 times, leading to a potential 518,400 re-executions. Even with the substitution

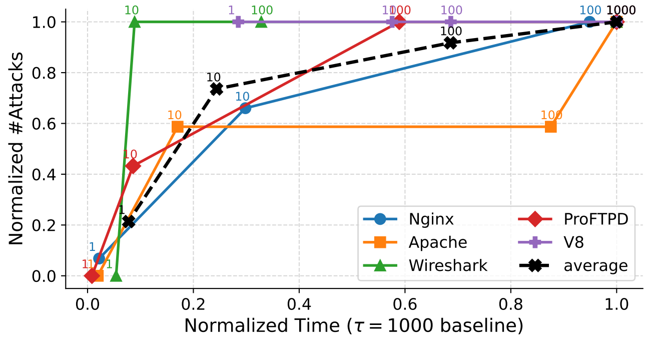


Fig. 6: Impact of substitution threshold. We set the threshold to 1,000, 100, 10, and 1, and measure the attack counts and runtimes.

threshold set to 1,000, SACK still performs 50,562 attempts, the highest among all evaluated programs.

2) Threshold τ of Substitutions: To quantify the efficiency gains and potential coverage loss introduced by limiting repetitive substitutions (§V-C1), we evaluate four thresholds, *i.e.*, 1,000, 100, 10, and 1, across all program oracles except SQLite3 and Sudo, whose executions complete within a few seconds. For each threshold, we measure the percentage of successfully constructed attacks and the end-to-end runtime, using $\tau = 1,000$ as the baseline. The results are illustrated in Figure 6. With $\tau = 100$, SACK reproduces all attacks in Nginx, ProFTPD, Wireshark and V8, and 58.7% attacks in Apache. Runtime decreases by 5.1%, 41.1%, 12.4%, 67.2% and 31.3% for Nginx, ProFTPD, Apache, Wireshark and V8, respectively. With $\tau = 10$, SACK still generates 66.0% Nginx attacks, 43.2% in ProFTPD, 58.7% in Apache, and all attacks in Wireshark and V8, while runtime drops by 70.2%, 80.3%, 91.4%, 91.1% and 42.4%. At the most aggressive setting ($\tau = 1$), SACK constructs only 17 attacks in Nginx and 1 in V8, with no attacks produced for other programs. Overall, the results show a clear trade-off between exploration cost and attack coverage. On average, using a threshold of $\tau = 100$ retains 91.7% of all attacks while requiring only 68.6% of the original execution time. In contrast, $\tau = 10$ further reduces the runtime to 24.4% of the baseline but preserves 73.6% of attacks. These results indicate that $\tau = 100$ is preferable when maximizing coverage is the priority, whereas $\tau = 10$ provides a more cost-efficient configuration when execution cost is the primary concern.

D. Incorporating Statically Inferred Targets

By default, SACK relies on dynamically collected targets to construct SUB attacks. However, the generic design of SACK can be easily extended to construct attacks using statically inferred, over-approximated target sets. As discussed in §IV-A, these over-approximated targets do not guarantee successful exploits under a precise CFI policy, but they provide a useful upper bound on the potential attack space. To explore this broader space, we incorporate two over-approximated target sets derived from MLTA [60] and TFA [58]. MLTA employs multi-layer type analysis to infer permissible targets, while TFA augments MLTA with data-flow analysis to improve precision. We use the inferred target sets both as substitution candidates and for subsequent CFI validation. Because MLTA and TFA are known to introduce false negatives [51], some dynamically observed targets may be missing from their results. To ensure completeness in our evaluation, we supplement each inferred set with any missed dynamic targets and run SACK on all evaluated programs using the existing oracles.

Table III reports the number of successful attacks constructed using over-approximated target sets. Incorporating TFA targets increases the number of attacks by 285 in Nginx, 237 in SQLite3, 32 in Sudo, 256 in Apache, and 153 in Wireshark. Using MLTA expands these counts to 331, 301, 36, 713, and 164, respectively. The growth can be substantial: for example, under Apache-A3, SACK discovers 22.5× more attacks using TFA and 45.3× more using MLTA. For some oracles, however, over-approximated targets do not lead to additional attacks. SQLite-Q1 shows no increase because its dynamic targets collected using the official test suite already achieve high coverage. V8-V1 also yields no additional attacks, as the exploit depends on a single specific substitution. For oracles that produce no exploits under dynamically collected targets, over-approximated target sets likewise do not uncover new attacks.

The increase in attack count comes with significant computational overhead. As shown in Table III, using TFA targets increases substitution time by 1.2×-14.5× across programs, while MLTA increases it by 1.7×-19.9×. The most extreme case arises in Wireshark-W1, where both analyses report over 40,000 potential targets for a single indirect call. To keep the experiments tractable, we reduce the substitution threshold to 10. Even so, evaluating Wireshark-W1 requires approximately 1.4 days with TFA targets and 2.7 days with MLTA targets. Moreover, TFA's static analysis phase alone takes 9.2 hours on Wireshark, whereas MLTA and TFA both complete within minutes for the remaining programs. These results demonstrate that over-approximated target sets can induce substantial overhead, and that threshold-based heuristics are essential for maintaining practical performance.

Taken together, these observations suggest a practical workflow for exploring SUB attacks. Under a precise static CFI policy, dynamically collected targets offer an efficient lower-bound estimate of the attack surface, while over-approximated targets serve as an upper-bound estimate when broader exploration is desired. For concrete CFI deployments, SACK can directly incorporate the policy's own allowed-target sets, enabling systematic and policy-compliant construction of SUB attacks across a wide range of applications.

VII. DISCUSSION

A. Oracle Reliability and Fallback Mechanisms

SACK's oracle constructor relies on LLMs and program official documentation. In practice, this approach is effective because many common targets of security research, CVE reports, and bug-bounty investigations are mature, widely deployed systems with extensive publicly available documentation. Examples include operating systems such as Linux man-pages and Windows API; major browsers such as Google Chrome docs, Mozilla Firefox docs, and WebKit docs; and widely used document-processing software such as Adobe

Acrobat SDK docs. These documents provide detailed setup, configuration, and API manuals to support large developer and user communities, enabling LLMs to synthesize correct and complete oracles. When manual adjustments are needed, they tend to be small, localized refinements, resolving ambiguity rather than correcting fundamental logic, and do not affect the overall scalability of the approach, as shown in Table II.

While many security-sensitive programs provide rich documentation, certain targets may expose only limited usage information, like smaller utilities or specialized components. In such cases, SACK can fall back on documentation-independent correctness signals. First, SACK may treat arbitrary code execution as the success condition [11], [26], marking an attack successful once unintended execution is observed. Second, SACK can monitor the invocation of critical system calls [86], [82], [12], [46], such as `execve` (arbitrary binary execution) or `seteuid` (privilege escalation), which are widely recognized as strong indicators of exploitation. Third, fallback oracles may track access to privileged resources [11], [5], including unauthorized file reads or unexpected exposure of sensitive memory. These mechanisms provide correctness signals independent of program documentation, making SACK effective when LLM-generated oracles cannot be fully constructed.

B. Support COTS Binaries

Although the current SACK implementation requires program source code for LLVM-based instrumentation, this is not a fundamental limitation. We can extend SACK to support COTS binaries through two strategies. First, we can instrument the program binary directly using binary-rewriting frameworks [23], [25], [15]. For example, RetroWrite [23] disassembles a binary, inserts instrumentation at the assembly level, and reassembles a new executable; E9Patch [25] injects instrumentation logic via trampoline redirection without altering the original control flow. SACK can adopt these techniques to perform target collection and substitution directly at the binary level. The second approach is to lift binaries into an intermediate representation suitable for analysis and instrumentation [76], [85], [22]. For example, McSema [22] and llvm-mctoll [85] translate binaries to LLVM IR, allowing SACK to reuse its LLVM-based instrumentation pipeline for COTS binaries. These techniques enable SACK to operate naturally on closed-source programs without requiring fundamental changes to its design.

C. SUB *Attacks under Dynamic CFI*

Dynamic CFI mechanisms incorporate runtime information to further constrain the valid target sets for ICT instructions [67], [81], [19], [24], [31], [38], [39]. They enforce more strict policies than static CFI techniques. Our work focuses on constructing SUB attacks under static CFI, where evading dynamic CFI is outside the scope of this study. Nevertheless, to contextualize the capability of SUB attacks, we briefly examine how such attacks interact with dynamic CFI mechanisms.

Depending on its enforcement strength, dynamic CFI can block a portion or all of the SUB attacks constructed under static CFI. For example, πCFI [67] admits an edge only after the corresponding function address has been taken during the current execution; it allows substitutions only to dynamically activated targets and rejects substitutions to any target not observed yet in the same run. Trace-based CFI schemes leverage hardware mechanisms such as Intel PT or LBR to record recent execution context (*e.g.*, call stack or short execution path) [24], [31], [38], [81], and validate each ICT against the corresponding context. Under such policies, a substitution is accepted only if it preserves the expected dynamic context. At the strongest end of the spectrum, μCFI [39] performs online points-to analysis and reduces every ICT to exactly one valid target. With this unique-target guarantee, SUB attacks are not infeasible anymore because no permissible substitutions exist.

While fully precise dynamic CFI mechanisms such as μCFI prevent all SUB attacks, approximate or incomplete schemes remain susceptible and we can adapt SACK to construct compatible evasions. For πCFI, instead of aggregating targets across multiple runs, SACK can restrict substitution candidates to the set of addresses observed as taken in the current execution. This ensures that all constructed substitutions remain consistent with πCFI's dynamic address-taken rule and therefore bypass its enforcement if a viable substitution exists. For trace-based CFI, SACK's target-collection phase can be extended to record contextual information such as call stacks, Intel PT traces, or LBR windows associated with each dynamic ICT. During exploit construction, SACK can synthesize both the substituted target and the requisite contextual pattern, enabling the generation of history-conforming or trace-conforming executions. In this way, SACK provides a principled foundation for evaluating and crafting evasions against dynamic CFI mechanisms whose enforcement is not fully precise.

D. Attack Coverage

Current conservative design choices of SACK likely underestimate the practical prevalence of SUB attacks. We identify several key factors contributing to this underestimation to motivate future research toward broader attack coverage.

We summarize four design choices in SACK that may lead to missed attacks: incomplete feature coverage, limited input diversity, omission of argument manipulation, and conservative verification. First, SACK focuses on explicit security features with clear observable consequences. However, this potentially overlooks implicit security features that attackers could still exploit or chain with other exploit primitives. Second, we use only a small set of inputs for each oracle, merely to test basic functionalities when official test cases are unavailable. Consequently, the attack engine exercises only a limited portion of the program, potentially missing indirect calls and code paths necessary for more complex attacks. Third, SACK performs function substitution without modifying function arguments. However, substituted functions may interpret the original arguments differently, leading to crashes or hangs. For example, 18.11% of substitution attempts in Nginx fail due to such mismatches. Incorporating argument adaptation could significantly expand the attack space. At last, SACK relies solely on dynamically collected targets to detect CFI violations. This may lead to false negatives, where viable attacks are discarded simply due to targets outside incomplete target collection.

Nevertheless, 419 attacks sufficiently demonstrate the wide applicability and systematic constructibility of SUB attacks.

E. Challenges of Accurate Behavior Monitoring

The implementation of behavior monitors may affect attack detection accuracy and efficiency. The reason is that security violations often manifest in subtle, context-dependent ways, influenced by side effects, environmental conditions, and timing behaviors. Even when the LLM correctly identifies the appropriate metrics for assessing feature enforcement, capturing all violations from a single perspective remains challenging. In our experiments, we identify 30 false positives, all attributable to implementation inaccuracies of behavior monitors.

For example, for Nginx-N2 that tests the rate-limiting feature, we expect Nginx to accept the first request and reject the second. The LLM accurately suggests inspecting HTTP status codes to verify the enforcement. Our initial monitor follows a client-side, output-based approach, which analyzes the responses of curl requests and expects a `200` OK followed by a `503` Service Unavailable. However, several substitutions cause Nginx to crash or hang, necessitating a timeout mechanism on the client side to infer server behavior. However, this mechanism slows down our testing process. To improve efficiency, we develop a server-side, log-based monitor that inspects Nginx log files for HTTP status codes. This provides a faster and more direct way to determine whether requests were accepted or rejected. Despite this improvement, certain cases remain ambiguous. For instance, the server may log two accepted requests, suggesting a potential bypass, but then abruptly terminate the connection without sending content to the client. Since the client does not receive the response, we count such attacks as false positives.

Fully resolving this issue would require a comprehensive understanding of the HTTP processing logic and the development of combined client-side and server-side monitors, which requires non-trivial efforts. In our evaluation, we adopt the server-side log-based monitor as all false positives stem from similar problems and represent a relatively low rate (6.7%).

VIII. CONCLUSION

In this paper, we present SACK, a systematic framework for constructing function substitution (SUB) attacks. By combining runtime-collected indirect call targets with LLM-synthesized security oracles, SACK enables scalable and automated generation of realistic, context-aware attacks. Our evaluation on seven widely used applications produces 419 successful SUB attacks. This result demonstrates that SUB attacks are practical, widespread and can be systematically constructed at scale.

ACKNOWLEDGMENT

We thank all anonymous reviewers for their insightful comments and valuable feedback. This research was supported by National Science Foundation (NSF) under grants CNS-2247652 and CNS-2339848. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

ETHICS CONSIDERATIONS

This study presents no ethical risks. First, our work does not involve the discovery of new vulnerabilities or the request for new CVEs. CVEs are designated for newly identified vulnerabilities, whereas our work focuses on exploit construction rather than vulnerability discovery. Our objective is to demonstrate that SUB attacks are practical, widespread, and systematically constructible; thus, identifying new vulnerabilities falls outside the scope of this research. Second, while we construct five real-world SUB attacks, all based on historical vulnerabilities (CVE-2017-6983, CVE-2021-30632 and CVE-2013-2028) that have been patched for several years. We reintroduce these bugs in isolated, controlled environments solely for demonstration purposes, ensuring no risk to real-world systems. Third, we did not report our findings to the developers of the evaluated applications. While developers can patch individual bugs (which have been patched in our case), preventing systematic exploitation requires strengthening general-purpose defenses such as control-flow integrity (CFI) [1] and code-pointer integrity (CPI) [49]. Fourth, our methodology follows standard practice established in prior work [11], [26], [4], [40], [44], [16], [33], [21], [12], [73], [75], which similarly evaluate exploit techniques using previously patched vulnerabilities or public proof-of-concept exploits.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-Flow Integrity,” in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005, pp. 340–353.

[2] Apache Software Foundation, “Apache HTTP Server Test Framework,” https://httpd.apache.org/test/, last accessed: 2025-07-21.

[3] ——, “Security Tips,” https://httpd.apache.org/docs/2.4/misc/security_tips.html, last accessed: 2025-07-05.

[4] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, “Automatic Exploit Generation,” in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011, pp. 74–84.

[5] E. Avllazagaj, Y. Kwon, and T. Dumitraș, “SCAVY: Automated Discovery of Memory Corruption Targets in Linux Kernel for Privilege Escalation,” in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, Philadelphia, PA, USA, Aug. 2024, pp. 7141–7158.

[6] L. Binosi, G. Barzasi, M. Carminati, S. Zanero, and M. Polino, “The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations,” in *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*, Salt Lake City, UT, USA, Nov. 2024, pp. 1360–1374.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-Oriented Programming: A New Class of Code-reuse Attack,” in *Proceedings of the 6th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, Hong Kong, China, Mar. 2011, pp. 30–40.

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based Greybox Fuzzing as Markov Chain,” in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016, pp. 1032–1043.

[9] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, “Control-Flow Integrity: Precision, Security, and Performance,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 16, 2017.

[10] Y. Cai, Y. Jin, and C. Zhang, “Unleashing the Power of Type-Based Call Graph Construction by Using Regional Pointer Information,” in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, Philadelphia, PA, USA, Aug. 2024, pp. 1383–1400.

[11] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, Washington, DC, Aug. 2015, pp. 952–963.

[12] N. Carlini and D. Wagner, "ROP is Still Dangerous: Breaking Modern Defenses," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, San Diego, CA, Aug. 2014, pp. 385–399.

[13] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-Flow Analysis," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, Boston, MA, USA, Aug. 2022, pp. 2531–2548.

[14] J. Chen, J. Wang, C. Song, and H. Yin, "Jigsaw: Efficient and Scalable Path Constraints Fuzzing," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, May 2022, pp. 18–35.

[15] P. Chen, Y. Xie, Y. Lyu, Y. Wang, and H. Chen, "Hopper: Interpretative fuzzing for libraries," in *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Copenhagen, Denmark, Nov. 2023, pp. 1600–1614.

[16] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium (USENIX Security)*, Baltimore, MD, Aug. 2005, pp. 177–191.

[17] Y. Chen, L. Xing, Y. Qin, X. Liao, X. Wang, K. Chen, and W. Zou, "Devils in the Guidance: Predicting Logic vulnerabilities in Payment Syndication Services through Automated Documentation Analysis," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, Santa Clara, CA, USA, Aug. 2019, pp. 747–764.

[18] B. Cheng, C. Zhang, K. Wang, L. Shi, Y. Liu, H. Wang, Y. Guo, D. Li, and X. Chen, "Semantic-enhanced Indirect Call Analysis with Large Language Models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sacramento, CA, USA, Oct.–Nov. 2024, pp. 430–442.

[19] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A Generic and Practical Approach for Defending against ROP Attack," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[20] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow stacks and Stack Canaries," in *Proceedings of the 10th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, Singapore, Singapore, Apr. 2015, pp. 555–566.

[21] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, San Diego, CA, Aug. 2014, pp. 401–416.

[22] A. Dinaburg and A. Ruef, "McSema: Static Translation of x86 Instructions to LLVM," in *ReCon 2014 Conference, Montreal, Canada*, 2014.

[23] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, Virtually, May 2020, pp. 1497–1511.

[24] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient Protection of Path-Sensitive Control Security," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, Vancouver, BC, Canada, Aug. 2017, pp. 131–148.

[25] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary Rewriting without Control Flow Recovery," in *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Virtual, Jun. 2020, pp. 151–163.

[26] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-grained Control Flow Integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015, pp. 901–913.

[27] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking Branch Predictors to Bypass ASLR," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[28] S. Feng, Z. Zhou, and K. Yang, "Many Birds, One Stone: Exploiting a Single SQLite Vulnerability Across Multiple Software," https://www.blackhat.com/docs/us-17/wednesday/us-17-Feng-Many-Birds-One-Stone-Exploiting-A-Single-SQLite-Vulnerability-Across-Multiple-Software.pdf, Las Vegas, NY, Jul. 2017, Black Hat USA.

[29] X. Feng, X. Liao, X. Wang, H. Wang, Q. Li, K. Yang, H. Zhu, and L. Sun, "Understanding and Securing Device Vulnerabilities through Automated Bug Report Analysis," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, Santa Clara, CA, USA, Aug. 2019, pp. 887–903.

[30] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, Virtual, Aug. 2020, pp. 10–10.

[31] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding Control Flows Using Intel Processor Trace," in *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

[32] M. Goddard, "The EU General Data Protection Regulation (GDPR): European Regulation That Has A Global Impact," *International Journal of Market Research*, vol. 59, no. 6, pp. 703–705, 2017.

[33] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, May 2014, pp. 575–589.

[34] Google, "Android Control Flow Integrity," https://source.android.com/devices/tech/debug/cfi, 2018.

[35] ——, "Chromium Control Flow Integrity," https://www.chromium.org/developers/testing/control-flow-integrity, 2022.

[36] ——, "Honggfuzz," https://google.github.io/honggfuzz/, 2023.

[37] J. Grossklags, C. Eckert, and Z. Lin, "τCFI: Type-assisted Control Flow Integrity for x86-64 Binaries," in *Proceedings of the 21st International Symposium of Research in Attacks, Intrusions, and Defenses (RAID)*, vol. 11050, Sep. 2018, p. 423.

[38] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent Backward-Edge Control Flow Violation Detection using Intel Processor Trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 173–184.

[39] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018, pp. 1470–1486.

[40] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, May 2016, pp. 969–986.

[41] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, May 2013, pp. 191–205.

[42] N. Inc., "Nginx Security Controls," https://docs.nginx.com/nginx/admin-guide/security-controls/, 2025.

[43] Intel Corporation, "Complex Shadow-Stack Updates (Intel® Control-Flow Enforcement Technology)," https://www.intel.com/content/www/us/en/content-details/785687/complex-shadow-stack-updates-intel-control-flow-enforcement-technology.html, 2023, accessed: 2025-07-05.

[44] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block Oriented Programming: Automating Data-Only Attacks," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018, pp. 1868–1882.

[45] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016, pp. 380–392.

[46] B. Johannesmeyer, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Data-Only Attack Generation," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, Philadelphia, PA, USA, Aug. 2024, pp. 1401–1418.

[47] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining Indirect Call Targets at the Binary Level," in *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*, Virtually, Feb. 2021.

[48] A. Kong, S. Zhao, H. Chen, Q. Li, Y. Qin, R. Sun, X. Zhou, E. Wang, and X. Dong, "Better Zero-Shot Reasoning with Role-Play Prompting," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HIT) (Volume 1: Long Papers)*, 2024, pp. 4099–4113.

[49] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014, pp. 147–163.

[50] C. Lattner, A. Lenharth, and V. Adve, "Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 278–289, 2007.

[51] G. Li, M. Sridharan, and Z. Qian, "Redefining Indirect Call Analysis with KallGraph," in *Proceedings of the 46th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, May 2025, pp. 2957–2975.

[52] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, USA, Nov. 2020, pp. 1821–1835.

[53] X. Liao, K. Yuan, X. Wang, Z. Li, L. Xing, and R. Beyah, "Acing the IOC Game: Toward Automatic Discovery and Analysis of Open-Source Cyber Threat Intelligence," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016, pp. 755–766.

[54] LIEF Developers, "LIEF Documentation," https://lief.re/doc/latest/index.html, 2025, accessed: 2025-07-26.

[55] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "PACStack: an Authenticated Call Stack," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, Virtually, Aug. 2021, pp. 357–374.

[56] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li, "GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, May 2022, pp. 2078–2095.

[57] Z. Lin, J. Li, B. Li, H. Ma, D. Gao, and J. Ma, "Typesqueezer: When Static Recovery of Function Signatures for Binary Executables Meets Dynamic Analysis," in *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Copenhagen, Denmark, Nov. 2023, pp. 2725–2739.

[58] D. Liu, S. Ji, K. Lu, and Q. He, "Improving Indirect-Call Analysis in LLVM with Type and Data-Flow Co-Analysis," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, Philadelphia, PA, USA, Aug. 2024, pp. 5895–5912.

[59] LLVM, "LibFuzzer - A Library for Coverage-guided Fuzz Testing," http://llvm.org/docs/LibFuzzer.html, 2023.

[60] K. Lu and H. Hu, "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019, pp. 1867–1881.

[61] Microsoft, "Data Execution Prevention (DEP)," May 2023, https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[62] Microsoft Corporation, "Control Flow Guard," https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2016.

[63] W.-L. Mow, S.-K. Huang, and H.-C. Hsiao, "LAEG: Leak-Based AEG Using Dynamic Binary Analysis to Defeat ASLR," in *Proceedings of the 2022 IEEE Conference on Dependable and Secure Computing (DSC)*. IEEE, 2022, pp. 1–8.

[64] Nergal, "The Advanced Return-into-lib(c) Exploits (PaX Case Study)," http://phrack.org/issues/58/4.html, Dec. 2001, Phrack.

[65] Nginx Inc., "Test Suite for Nginx," https://github.com/nginx/nginx-tests, accessed: 2025-07-21.

[66] B. Niu and G. Tan, "Modular Control-Flow Integrity," in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014, pp. 577–587.

[67] ——, "Per-Input Control-Flow Integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015, pp. 914–926.

[68] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking Holes in Information Hiding," in *25th USENIX Security Symposium (USENIX Security)*, 2016, pp. 121–138.

[69] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards Automating Risk Assessment of Mobile Applications," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, Washington, DC, Aug. 2013, pp. 527–542.

[70] B. V. Patel, "A Technical Look at Intel's Control-flow Enforcement Technology," https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html, 2020.

[71] PaX Team, "PaX Address Space Layout Randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[72] Z. Qi, J. Hu, Z. Xiao, and H. Yin, "SymFit: Making the Common (Concrete) Case Fast for Binary-Code Concolic Execution," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, Philadelphia, PA, USA, Aug. 2024, pp. 415–432.

[73] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, May 2015, pp. 745–762.

[74] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *Proceedings of the 20th USENIX Security Symposium (USENIX Security)*, San Francisco, CA, Aug. 2011.

[75] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2007, pp. 552–561.

[76] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok: (State of) the Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, May 2016.

[77] SQLite, "How SQLite Is Tested," https://www.sqlite.org/testing.html, May 2022.

[78] ——, "Defense against The Dark Arts," https://www.sqlite.org/security.html, 2025.

[79] ——, "The –safe Command-line Option," https://sqlite.org/cli.html#the_safe_command_line_option, 2025 (last visited).

[80] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, San Diego, CA, Aug. 2014, pp. 941–955.

[81] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015, pp. 927–940.

[82] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017, p. 1675–1689.

[83] V. van der Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, May 2016, pp. 934–953.

[84] T. Xia, H. Hu, and D. Wu, "DEEPTYPE: Refining Indirect Call Targets with Strong Multi-layer Type Analysis," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, Philadelphia, PA, USA, Aug. 2024, pp. 5877–5894.

[85] S. B. Yadavalli and A. Smith, "Raising binaries to llvm ir with mctoll (wip paper)," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2019, pp. 213–218.

[86] H. Ye, S. Liu, Z. Zhang, and H. Hu, "VIPER: Spotting Syscall-Guard Variables for Data-Only Attacks," in *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*, Anaheim, CA, USA, Aug. 2023, pp. 1397–1414.

[87] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based Automatic generation of Proof-of-Concept Exploits," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017, pp. 2139–2154.

[88] M. Zalewski, "New in AFL: Persistent Mode," https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html.

[89] ——, "American Fuzzy Lop (2.52b)," http://lcamtuf.coredump.cx/afl, Nov. 2017.

[90] H. Zhang, J. Liu, J. Lu, S. Chen, T. Han, B. Zhang, and X. Gong, "Reviving Discarded Vulnerabilities: Exploiting Previously Unexploitable Linux Kernel Bugs Through Control Metadata Fields," in *Proceedings of the 32nd ACM Conference on Computer and Communications Security (CCS)*, Taipei, Taiwan, Oct. 2024.

[91] M. Zhang and R. Sekar, “Control Flow Integrity for COTS Binaries,” in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, Washington, DC, Aug. 2015, pp. 337–352.

[92] W. Zhu, Z. Feng, Z. Zhang, J. Chen, Z. Ou, M. Yang, and C. Zhang, “Callee: Recovering Call Graphs for Binaries with Transfer and Contrastive Learning,” in *Proceedings of the 44th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, May 2023, pp. 2357–2374.

[93] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, “Fuzzing: A Survey for Roadmap,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[94] Z. Zhu and T. Dumitraş, “FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature,” in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016, pp. 767–778.

[95] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, “SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux Kernel,” in *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, Boston, MA, USA, Aug. 2022, pp. 3201–3217.

APPENDIX A
ARTIFACT APPENDIX

This Artifact Appendix summarizes the artifact components and the steps required to reproduce our key results.

A. Description & Requirements

This artifact includes materials to reproduce the results presented in §VI-A, §VI-B, and §VI-C of the paper.

1) How to access: The artifact is publicly available at this link and citable via the DOI https://doi.org/10.5281/zenodo.17782315. The repository contains components of the SACK framework along with detailed instructions. It also contains the original experimental data used in our evaluation. Additionally, the online repository and Zenodo provide four pre-built Docker images, each containing the necessary pre-built target programs.

2) Hardware dependencies: Reproducing all experiments requires at least 60 GB of free disk space for Docker images; evaluating only a subset requires proportionally less.

3) Software dependencies: The artifact requires Docker to load and run the Docker images. An active internet connection is needed to download third-party packages and perform LLM queries. Oracle generation is based on GPT-4.1; therefore, reproducing this process requires a valid API key.

Our experiments were conducted on a 64-bit Ubuntu 22.04 host system using Docker 28.5.2 (build ecc6942). Comparable environments and versions should work as well.

4) Benchmarks: None.

B. Artifact Installation & Configuration

1) Select Docker Image(s): Choose one or more Docker images depending on the oracles you intend to evaluate. (See `README.md` for details.)

2) Prepare a Docker Image: (e.g., `sack_main`)

```
apt-get install -y skopeo
skopeo copy \
    oci-archive:sack_main_latest.oci.tar \
    docker-archive:sack_main_latest.docker.tar
docker load -i sack_main_latest.docker.tar
# Example output:
#   Loaded image ID: sha256:<IMAGE_ID>
docker tag sha256:<IMAGE_ID> sack_main:latest
```

3) Start the Container: Start the container with the required permissions:

```
docker run --cap-add=SYS_PTRACE \
           --name ae-sack-main \
           --security-opt seccomp=unconfined \
           -it sack_main:latest \
           tail -f /dev/null
docker exec -it ae-sack-main /bin/bash
```

4) Run the Experiments: Follow the instructions in the corresponding component directory to run the experiments.

C. Experiment Workflow

We provide a set of scripts and detailed guidelines to demonstrate the functionality of our prototype and the reproducibility of its results. The evaluation consists of three core components: the oracle constructor, the target collector, and the attack engine. The oracle constructor demonstrates our pipeline's ability to query LLMs and generate security oracles. The target collector illustrates the method used to identify and collect sub-ground truth indirect call targets for substitution. The attack engine integrates the outputs from both the oracle constructor and the target collector to automatically discover SUB attacks.

For each target oracle, the corresponding scripts can be used to evaluate any or all of the three components. The results can be compared against the values reported in the paper (e.g., the number of successful attacks) or against the original experimental data provided in the artifact package (e.g., the actual generated attacks). More comprehensive instructions, along with working examples, are available in the repository's README file and in the documentation provided for each individual component.

D. Major Claims

Across all experiments, the exact numbers may vary slightly from those reported in the paper, but the overall conclusions are expected to remain consistent.

- (C1) Oracle constructor: The oracle constructor is capable of producing security oracles for the target programs used in our evaluation. This is demonstrated in Experiment E1, with the generated oracles selected and listed in Table I.
- (C2) Target collector: The target collector dynamically collects indirect call targets during benign executions, serving as sub-ground truth for function substitution attacks. This is demonstrated in Experiment E2, following the same procedure described in §IV-A and §V-B.
- (C3) Attack engine: The attack engine can automatically construct end-to-end SUB attacks. This capability is demonstrated in Experiment E3, with attack counts comparable to those reported in Table I and Table III.

Additionally, the repository includes the original generated oracles, collected sub-ground truth targets, and the discovered concrete attacks as a reference for the above experiments.

E. Evaluation

In this section, we outline the steps required to run the experiments and reproduce the results presented in the paper. Please refer to the main README file in the repository, as well as the README files within each component directory, for detailed instructions.

Our evaluation is organized on a per-program basis. We recommend selecting a subset of target programs for testing. The estimated execution times listed below indicate the time required to evaluate the selected Nginx oracle. The listed paths indicate their locations within the Docker image or relative paths in the online repository.

1) Experiment (E1): [Oracle constructor] [5 human-minutes + 8 compute-minutes]: Refer to the README in `/ae-sack/oracle-generation/`. This experiment evaluates the oracle generation capability. The code queries the LLM for security features, crawls and preprocesses documentation, applies two filters, and then performs a second query to generate security oracles.

[Preparation] Navigate to `/ae-sack/oracle-generation/oracle_generation_nginx`.

[Execution] Follow the instructions provided in the README file within the folder.

[Results]

The generated oracles are located in `./oracle_generation_with_provided_feature/` and can be compared with the reference oracles in `./metadata/`. The results should be similar. The evaluation features listed in the README should also be present in the generated oracles.

2) Experiment (E2): [Target collector] [5 human-minutes + 1 compute-minutes]: Refer to the README in `/ae-sack/target-collector/`. This experiment evaluates the collection of sub-ground truth indirect call targets.

[Preparation] Use image `sack_main` to create a container and navigate to `/target/nginx/nginx-basic-auth/bin/sbin/`.

[Execution] Follow the instructions in `/ae-sack/target-collector/README.md` for collecting the `n1_auth` sub-ground truth.

[Results] Refer to the `Verification Method` section in `README.md`. The collected indirect call targets can be inspected and compared, and can further be used to reproduce the attacks demonstrated in Experiment E3.

3) Experiment (E3): [Attack engine] [2 human-minutes + 10 compute-minutes]: Refer to the README in `/ae-sack/scripts/`. This experiment evaluates the attack generation of SACK.

[Preparation] Use image `sack_main` to create a container and navigate to `/target/nginx/nginx-basic-auth/bin/sbin/`.

[Execution]

1) Run the attack engine by following the instructions in `/ae-sack/scripts/nginx/n1_auth/run.readme`.
2) After the substitution process completes, execute the analysis script `/ae-sack/scripts/nginx/n1_auth/analyze.sh` to collect the results.

[Results]

Refer to the `Result Verification` section of the README in `/ae-sack/scripts/`. The discovered attacks are located in `report_satisfied.txt` within the latest `result.*/` directory. They are also printed to `stdout`. You can verify that the A1 and A2 counts match those in Table III. Additionally, you may compare the concrete attacks with the original reference data in `/ae-sack/attack_metadata/`.