Anota: Identifying Business Logic Vulnerabilities via Annotation-Based Sanitization

Meng Wang*, Philipp Görz*, Joschua Schilling*, Keno Hassler*, Liwei Guo†, Thorsten Holz ‡, Ali Abbasi§
*§CISPA Helmholtz Center for Information Security, †University of Electronic Science and Technology,
‡Max Planck Institute for Security and Privacy
* first.last@cispa.de, † lwg@uestc.edu.cn, ‡thorsten.holz@mpi-sp.org, § abbasi@cispa.de

***Abstract*—Detecting business logic vulnerabilities is a critical challenge in software security. These flaws come from mistakes in an application's design or implementation and allow attackers to trigger unintended application behavior. Traditional fuzzing sanitizers for dynamic analysis excel at finding vulnerabilities related to memory safety violations but largely fail to detect business logic vulnerabilities, as these flaws require understanding application-specific semantic context. Recent attempts to infer this context, due to their reliance on heuristics and non-portable language features, are inherently brittle and incomplete. As business logic vulnerabilities constitute a majority (27 of the CWE Top 40) of the most dangerous software weaknesses in practice, this is a worrying blind spot of existing tools.**

In this paper, we tackle this challenge with ANOTA, a novel human-in-the-loop sanitizer framework. ANOTA introduces a lightweight, user-friendly annotation system that enables users to directly encode their domain-specific knowledge as lightweight annotations that define an application's intended behavior. A runtime execution monitor then observes program behavior, comparing it against the policies defined by the annotations, thereby identifying deviations that indicate vulnerabilities. To evaluate the effectiveness of ANOTA, we combine ANOTA with a state-of-the-art fuzzer and compare it against other popular bug finding methods compatible with the same targets. The results show that ANOTA+FUZZER outperforms them in terms of effectiveness. More specifically, ANOTA+FUZZER can successfully reproduce 43 known vulnerabilities, and discovered 22 previously unknown vulnerabilities (17 CVEs assigned) during the evaluation. These results demonstrate that ANOTA provides a practical and effective approach for uncovering complex business logic flaws often missed by traditional security testing techniques.

I. INTRODUCTION

Fuzzing, a dynamic program analysis technique, is widely used to discover software vulnerabilities across diverse systems. Fuzzing operates by supplying the Program Under Test (PUT) with a large number of malformed or unexpected inputs, aiming to trigger vulnerable states. However, the fuzzer itself is typically oblivious to whether a given input has successfully exposed a vulnerability. This is where sanitizers play an indispensable role. Sanitizers are instrumental tools that monitor program execution, detect predefined classes of errors, and report these violations to the fuzzer, thereby locating the vulnerability-inducing inputs [1]. Historically, sanitizers are developed for detecting memory-related [2]–[7], undefined behavior [8], and data race [9]–[11] vulnerabilities in memory-unsafe languages such as C and C++. More recently, adapting this technique to memory-safe languages such as Python and PHP has led to the development of sanitizers targeting a broader range of vulnerabilities. These include issues such as command injection, Server-Side Request Forgery (SSRF), and path traversal [12]–[21].

To examine the extend to which these recent sanitizers are capable of detecting the most prevalent vulnerabilities, we analyzed the 40 most dangerous software weaknesses (CWE Top 40) [22]. Despite recent progress in sanitizer development, we find that a considerable number of these critical vulnerabilities are either not addressed at all or inadequately addressed. This worrying observation forms the primary motivation for our work: there is a need to develop sanitizers to light up this detection blind spot.

As shown in Table I, our analysis of fuzzing sanitizers identifies two groups of weaknesses: *Unaddressed Weaknesses* and *Narrowly-Addressed Weaknesses*.

Unaddressed Weaknesses (blue rows in Table I): refers to weaknesses for which no sanitizer has been proposed yet. A prominent example is authorization: the sanitizer needs to know the application's context to identify whether the user has the privilege to access certain resources like reading/writing variables, calling functions, or executing privileged code. Similar obstacles block other CWE entries, such as Uncontrolled Search Path Element, for which the sanitizer needs to have knowledge of the developer's intended file system access privileges.

Narrowly-Addressed Weaknesses (gray rows in Table I): Even when sanitizers do exist for certain vulnerabilities, their solutions often have limited generalizability. Many established approaches [14], [18], [26], [27] exhibit language-specificity, depending on inferring program context from idiosyncratic language-specific features, thereby limiting their applicability to a single language, without offering paths to generalize the approach. Other techniques [16], [21], [31] impose strong pre-conditions that are often unattainable in practice, such as

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.2026.240938
www.ndss-symposium.org

TABLE I

ANOTA'S SUPPORT FOR THE CWE TOP 40 SECURITY WEAKNESSES. "✔" SHOWS SUPPORTED VULNERABILITY CLASSES, WHILE "🛠" REFERS TO VULNERABILITIES WHERE AN ANNOTATION SYSTEM IS NOT USEFUL, AS THEY CAN BE DIRECTLY PATCHED IF THE DEVELOPER HAS THE CORRECT INTUITION. FURTHERMORE, "📘" SHOWS CLASSES SUPPORTED BY COMPLIMENTARY TOOLS. THIS EXISTING WORK IS LISTED IN THE LAST COLUMN, EVEN THOUGH THESE TOOLS MOSTLY FOCUS ON TECHNICAL, NON-BUSINESS LOGIC-RELATED ISSUES. BUSINESS LOGIC NEEDED (BLN) REFERS TO WHETHER THE BUGS CAN ONLY BE FOUND WITH (●) OR STRICTLY WITHOUT (○) AN UNDERSTANDING OF THE APPLICATION'S BUSINESS LOGIC. THERE ARE VULNERABILITY CLASSES WHERE ONLY SOME BUGS REQUIRE THIS UNDERSTANDING (◐).

Rank	CWE	Description	Exploitation	BLN	ANOTA	Existing Work
1	79	Cross-site Scripting	APP Data	○	📘	FuzzOrigin [23], KameleonFuzz [24], webFuzz [19]
2	787	Out-of-bounds Write	Memory	○	📘	ASan [2]
3	89	SQL Injection	DataBase	○	📘	Witcher [12], Atropos [14]
4	352	Cross-Site Request Forgery (CSRF)	System Call	◐	🛠	WebFuzzAuto [20]
5	22	Path Traversal	System Call	●	✔	Atropos [14], PHUZZ [25]
6	125	Out-of-bounds Read	Memory	○	📘	ASan [2]
7	78	OS Command Injection	System Call	◐	✔	Witcher [12], Atropos [14]
8	416	Use After Free	Memory	○	📘	ASan [2]
9	862	Missing Authorization	Object Access/Code Exec.	●	✔	
10	434	Unrestricted Upload of File	System Call	◐	✔	Atropos [14], UFuzzer [18], URadar [26]
11	94	Code Injection	System Call/Data Flow	◐	✔	
12	20	Improper Input Validation	System Call	◐	🛠	Witcher [12], Atropos [14]
13	77	Command Injection	System Call	◐	✔	Witcher [12], Atropos [14]
14	287	Improper Authentication	Object Access/Code Exec.	●	✔	
15	269	Improper Privilege Management	Object Access/Code Exec.	●	✔	
16	502	Deserialization of Untrusted Data	System Call	●	✔	ODDFuzz [27], PHUZZ [25]
17	200	Exposure of Sensitive Information	Data Flow/System Call	●	✔	EDEFuzz [16], FLOWFUZZ [21]
18	863	Incorrect Authorization	Object Access/Code Exec.	●	✔	
19	918	Server-Side Request Forgery (SSRF)	System Call	◐	✔	Atropos [14], SSRFuzz [15]
20	119	Improper Restriction of Ops. w/i Mem. Buffer	Memory	○	📘	ASan [2]
21	476	NULL Pointer Dereference	Memory	○	📘	ASan [2]
22	798	Use of Hard-coded Credentials	Data	●	🛠	
23	190	Integer Overflow or Wraparound	Memory	○	📘	UBSan [8]
24	400	Uncontrolled Resource Consumption	DOS	○	✔	All Fuzzers
25	306	Missing Authentication for Critical Function	Object Access/Code Exec.	●	✔	
26	770	Allocation of Resources Without Limit	DOS	○	📘	All Fuzzers
27	668	Exposure of Resource to Wrong Sphere	Object Access/Code Exec.	●	✔	EDEFuzz [16]
28	74	Improper Neutralization of Special Elements	System Call/Data Flow	◐	✔	Witcher [12]
29	427	Uncontrolled Search Path Element	System Call	◐	✔	
30	639	Authorization Bypass	Object Access/Code Exec.	●	✔	
31	532	Insertion of Sensitive Information into Log File	Data Flow	●	✔	FLOWFUZZ [21]
32	732	Incorrect Permission Assignment	Object Access/Code Exec.	●	✔	
33	601	Open Redirect	System Call	◐	✔	OpenRedireX [28]
34	362	Race Condition	System Call/Object Access	◐	✔	TSan [9], CONZZER [29], krace [30]
35	522	Insufficiently Protected Credentials	Data Flow/Object Access	●	✔	
36	276	Incorrect Default Permissions	Object Access/Code Exec.	●	🛠	
37	203	Observable Discrepancy	Data Flow	●	✔	CT-Fuzz [31]
38	59	Link Following	System Call	○	🛠	
39	843	Type Confusion	Memory	○	📘	type-san [32]
40	312	Cleartext Storage of Sensitive Information	Data Flow	○	📘	

EDEFuzz [16], which requires a rendered web interface for its operation, a setup not always available for every application.

Our investigation into these detection gaps reveals a common cause: these weaknesses are overwhelmingly business logic vulnerabilities. Detecting such flaws requires a deep understanding of an application's specific rules and intended workflows: a level of understanding semantic context that automated tools struggle to achieve. To pursue full automation, existing tools are forced to approximate this understanding using predefined heuristic patterns of behavior that seem suspicious. However, these heuristics are selected artificially and cannot capture the nuanced, application-specific context required to reliably distinguish legitimate behavior from a true security violation.

Consider the example of a file upload feature. State-of-the-art tools like Atropos [14] attempt to detect business logic vulnerabilities like unrestricted file uploads using narrow, hard-coded heuristics. More specifically, Atropos instruments a predefined list of standard PHP functions and flags only PHP file uploads as malicious. This detection logic is inherently brittle and is easily bypassed if an attacker uploads a different dangerous file type or if the application uses a custom function not on the tool's predefined list. This reliance on superficial patterns fails against other context-dependent attacks, such as an authenticated user exploiting a path traversal flaw to upload a file into another user's private folder. Detecting such a violation requires specific contextual knowledge that the heuristic lacks: Who is the requesting user? What are their permissions for the target directory? For a fully automated tool, reliably inferring these fine-grained policies is fundamentally challenging; The pursuit of automation often comes at the expense of generalizability and accuracy. However, such semantic knowledge is readily available to an application's developer or user, who can easily define the intended policy.

In the research area of vulnerabilities in memory-unsafe programming languages, there has been a shift from early attempts to find memory errors solely by observing program crashes [33] towards sanitization. Tools like Address-

Sanitizer [2] detect the violation at its source (e.g., an illegal memory operation), thereby finding bugs that would not have manifested in observable crashes. This works well because the correct behavior is *implicitly* encoded for these vulnerabilities—a memory violation is never correct. This tool has been instrumental in wide-scale fuzz testing of open-source software, uncovering more than 36,000 security bugs [34]. We would like to benefit similarly from sanitization in the field of business-logic vulnerabilities. However, in this field, the correct behavior is not clear and has to be stated *explicitly* to overcome the limitations of narrow heuristics.

To this end, we introduce ANOTA, a novel human-in-the-loop sanitizer framework built with the classic systems principle [35] of separating policy from mechanism. Our approach empowers developers to encode their knowledge, avoiding the pitfalls of automated context inference. ANOTA is designed to work with any existing fuzzer, typically by reporting policy violations as program crashes. ANOTA consists of two core components: A lightweight, easy-to-use annotation system through which a developer or security analyst defines application-specific semantic rules (the policy), and a general-purpose, language-agnostic runtime policy monitor that enforces these policies during the program execution (the mechanism). This design fundamentally shifts the generalization effort. Instead of building increasingly complex and brittle heuristics, ANOTA provides an extensible framework for users to express their semantic understanding of the application's intended behavior. The human-in-the-loop methodology has proven highly effective in other challenging domains such as software verification and interactive machine learning [36]–[39]. The underlying idea is that humans, equipped with deep contextual understanding and knowledge of the system's business logic, can provide invaluable input when facilitated by an intuitive and effective generic API.

We implemented a prototype of ANOTA for Python applications, chosen for its widespread popularity [40] and the prevalence of complex applications built with it. To evaluate the effectiveness and overhead of ANOTA, we integrate it with the Python fuzzer Atheris [41] to create ANOTA+FUZZER. We evaluate ANOTA+FUZZER against an unmodified Atheris fuzzer, showing ANOTA could empower the baseline fuzzer to detect various business-logic vulnerabilities and achieve significantly higher precision and recall on our benchmark datasets. Furthermore, we validated ANOTA's practical utility by re-discovering 43 known vulnerabilities and uncovering 22 new vulnerabilities in popular, actively maintained Python projects. To date, 17 of these discoveries have been assigned CVE identifiers. An annotation study further confirmed that ANOTA's annotations are intuitive, and our performance evaluations show that it incurs minimal runtime overhead (about 10% in an artificial worst-case setup) while benchmarking using Python Performance Benchmark Suite [42].

Contributions. We make the following key contributions:

- We systematically analyze existing sanitizers against the CWE Top 40, demonstrating that the most critical unaddressed weaknesses are business logic vulnerabilities whose detection requires deep semantic context that current fuzzing sanitizers cannot capture.
- We propose a new sanitization paradigm based on the principle of separation of policy and mechanism. We design a lightweight annotation system that enables developers to formally express a program's intended behavior, thereby guiding dynamic analysis.
- We present the design and implementation of ANOTA, a sanitizer that instantiates our paradigm by integrating developer annotations with a runtime monitor to detect policy violations at their source.
- We implement ANOTA for Python and create ANOTA+FUZZER. Through comprehensive evaluation, we demonstrate ANOTA's effectiveness in finding real-world bugs, uncovering 22 impactful zero-day vulnerabilities, and outperforming state-of-the-art tools.

Following our commitment to open science, we make our implementation and evaluation scripts available online at https://github.com/ANOTA-Sanitizer/ANOTA.

II. BACKGROUND

We start by providing an overview of sanitizers developed to detect vulnerabilities listed in the CWE Top 40, then analyze the reason why certain vulnerabilities are Unaddressed or Narrowly-Addressed. We conclude with a motivating example to illustrate our approach.

A. Limitations of Existing Sanitizers

Sanitizers are a key component of dynamic analysis like fuzzing. They act as runtime oracles that monitor a program's execution to determine if its behavior indicates a security vulnerability. Sanitizers have evolved significantly. Early approaches often relied on simple signals like program crashes [33] to detect bugs. In contrast, modern tools like AddressSanitizer [2] are far more sophisticated, excelling at finding errors like memory corruption that violate universal rules of program execution.

However, this success does not fully extend to all the vulnerabilities prevalent in modern applications. Our analysis of sanitization capabilities against the CWE Top 40 vulnerabilities reveals a blind spot (detailed in Table I). This blind spot is most apparent for what we term *Unaddressed Weaknesses*. Many critical vulnerabilities, particularly those related to authorization and authentication, are invisible to existing sanitizers because detecting them requires application-specific context. For instance, to find an improper privilege management flaw, a sanitizer must understand the application's intended permission model to know if a user's access attempt is unauthorized, a level of semantic knowledge that generic tools lack.

In other cases, where sanitizers do exist, they are often *Narrowly-Addressed Weaknesses*. These tools attempt to infer context through methods that are brittle and impractical. Many rely on language-specific heuristics; for example, Atropos [14]

```python
1  def SafeURLOpener(input_link):
2    block_schemes = ["file", "php", "ftp", "data"]
3    block_host = ["youtube.com", "instagram.com"]
4    input_scheme = urlparse(input_link).scheme
5    input_hostname = urlparse(input_link).hostname
6    if input_scheme in block_schemes:
7      return
8    if input_hostname in block_host:
9      return
10   target = urllib.request.urlopen(input_link)
11   print(target.read())
```

Listing 1. Motivation example for our approach.

and PHUZZ [25] are tied to standard PHP functions. Atropos [14] also pre-defined a conservative heuristic as mentioned in the previous section. Neither method is portable, but both can be easily bypassed. ODDFuzz [27] illustrates this tight coupling with its design specifically tailored to sensitive call-sites within Java applications. Furthermore, other tools depend on restrictive operational pre-conditions, such as the GUI required by EDEFuzz [16] or the identical execution environments and significant manual setup needed by FLOWFUZZ [21].

All of these limitations stem from a single, fundamental challenge: the difficulty of automatically inferring deep, application-specific context. The repeated failure of automated heuristics to reliably capture this context motivates the need for an alternative approach, one that can directly equip sanitizers with the crucial awareness they currently lack. We propose that an intuitive and lightweight annotation framework, enabling users to systematically embed their domain-specific knowledge and security intent directly into the program, can grant sanitizers access to the precise contextual awareness they currently lack. Such an approach would circumvent the limitations of language-dependent heuristics and the complexities of inferring context from external interfaces, thereby paving the way for more precise, effective, and broadly applicable vulnerability detection.

B. Motivating Example

Consider the Python program in Listing 1. The developer wants to filter out certain network schemes and host names to avoid fetching content from remote user-provided URLs with such patterns. The developer imports the `urllib` library from Python's standard library for working with URLs. The developer defines schemes and host names to be blocked (lines 2–3) and relies on the `urlparse` API to parse the input URLs (lines 4–5). Then the code checks if the URL matches the block pattern (lines 6–9) to decide whether to retrieve the remote sites' contents (lines 10–11).

Although at first glance this implementation looks acceptable, it contains the vulnerability CVE-2023-24329, introduced by `urllib` in its API `urllib.parse.urlparse`. This API mistakenly parses URLs beginning with whitespace, which allows an attacker to bypass the filtering mechanism shown in Listing 1. For instance, if the remote user provides the URL `''https://youtube.com''`, the `SafeURLOpener` will block the request. However, if the URL is prefixed with whitespace (e.g., `''␣␣␣https://youtube.com''`), the blocking mechanism will be bypassed.

Detecting this vulnerability is challenging because it requires understanding application-specific business logic, not just common bug heuristics or patterns Static analysis fails, while it can identify the urlopen API call, it is blind to the custom filtering logic that defines the intended policy. Dynamic analysis like fuzzing also fails; although it can possibly trigger the fault, the violation goes undetected because it neither crashes the program nor violates a rule known to existing sanitizers. Ultimately, both approaches have the same core problem of lacking access to the application's context. Effective detection is only possible by comparing a program's runtime behavior against its application-specific semantic rules (the policy) to tell the difference between a flaw and correct execution.

III. Design

In the following, we present the design and implementation of ANOTA, a human-in-the-loop framework that enables a developer or security analyst to sanitize hard-to-detect business logic bugs using annotations. A high-level overview of ANOTA is shown in Figure 1. At its core, a policy monitoring module escalates policy violations to a crash that can be detected by fuzzing. Our approach consists of two phases: an initial annotation phase and an application testing phase. In the first phase, the user of ANOTA adds annotations to the code in step ❶. In the testing phase, a dynamic analysis tool like a fuzzer executes ❷ the annotated application with a series of test inputs. The instrumented execution environment parses the policies reflected in the annotations ❸ and the policy monitor enforces the policies and checks the execution of the program ❹. If a policy violation is detected during the testing process, it is included in the vulnerability report ❺. In the following, we present our design of the annotations and the policy monitor in more detail.

A. Annotations

The core design principle of the annotation system is to enable developers and security analysts to directly encode their contextual understanding of the program into a set of enforceable runtime policies. These policies are defined through annotations that grant or revoke permissions for program actions, offering fine-grained control, allowing developers to define policies for specific parts of an application using two models: a blocklist, which prohibits a defined set of actions while allowing all others, or a more restrictive allowlist, which permits only a defined set of actions while denying all others by default. Also, annotations can be controlled individually and toggled during runtime.

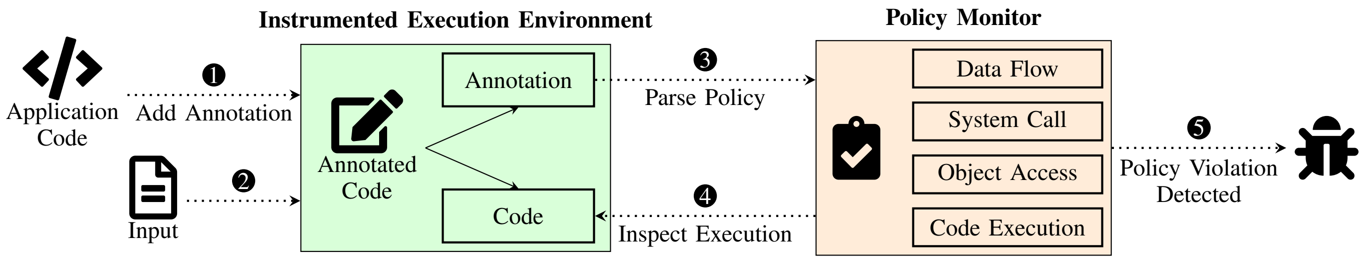

Fig. 1. High-level overview of ANOTA.

TABLE II
OVERVIEW OF ANNOTATIONS PROVIDED BY ANOTA

Type	Options	Example
System Call	Syscalls and args	① `SYSCALL.BLOCK('execv', 'execveat', 'execve')`
		② `SYSCALL.READ.BLOCK (PATH='/etc/')`
		③ `SYSCALL.EXECVE.ALLOW (PATH='/bin/ls')`
Data Flow	Sanitization, Sink	④ `TAINT(pwd, sanitization= [hash], Sink=[print])`
Object Access	Read, Write, Execute	⑤ `WATCH.ALLOW(admin_data,'r')`
		⑥ `WATCH.BLOCK(admin_api,'x')`
		⑦ `WATCH.CON(passwd_cmp)`
Code Execution	Condition	⑧ `EXECUTION.BLOCK (user.type!='admin')`
Annotation Clear	Annotation	⑨ `CLEAR(SYSCALL.EXECVE. ALLOW(PATH='ls'))`
		⑩ `SYSCALL.NETWORK.CLEAR()`

The annotations provided by ANOTA (summarized in Table II) are based on the runtime behavior caused by the CWE Top 40 vulnerabilities [22]. Our analysis of business logic vulnerabilities detailed in Table I, identified four critical types of deviant behavior: (1) unintended system call usage, (2) unintended code execution, (3) unintended variable access, and (4) unintended data flow. Consequently, we designed four primary annotation types to counter these behaviors, along with a fifth to disable annotations during runtime

As shown in Table I, our set of annotations was designed to be expressive enough to address the business logic vulnerabilities found in the CWE Top 40. The system is also extensible by design. Because we separate the annotation definitions (policy) from the runtime monitor (mechanism), users can add new annotation types for other vulnerabilities without re-engineering the core system. This typically just requires defining the new annotation's syntax and implementing a corresponding check in the runtime monitor, allowing the framework's detection capabilities to expand as new vulnerability classes emerge.

System Call annotations enforce access only to the allowed system calls and resources. Using `SYSCALL.[ALLOW|BLOCK](List of System Calls)` specifies which system call is allowed or blocked. For example, ① in Table II blocks three `execv`-related system calls. Additionally, more fine-grained policies can restrict specified system calls to specific resources using the syntax: `SYSCALL.[SYSCALL NAME|SYSCALL CLASS].[ALLOW|BLOCK](arguments)`. As shown in ③, this allows a user to define that the `/etc/` directory should not be accessible by the `read` system call in ②, or the `execve` system call can only use the `/bin/ls` executable.

To make policy writing more intuitive for users, ANOTA provides a simplified syntax ("syntactic sugar") for common system calls. For example, `FILE` can be used to control file/directory, attribute modification, and read/write permissions for all file-accessing system calls at once. Similarly, `NETWORK` represents all network-related system calls, and options like `SCHEME` are provided to control the URL scheme. Note that a more knowledgeable user could also directly define raw policies to control each argument of a system call for even more fine-grained control. Additionally, wildcards can be used to define matches similarly to shell expansion, e.g., `SYSCALL.READ.BLOCK(PATH='/upload_folder/*.php')` blocks read access to all PHP scripts in `upload_folder`.

Data Flow annotations instruct ANOTA to track the data flow of a variable. The `TAINT(Target_Variable, Sanitization Method, Sink)` annotation specifies which variable(s) to taint. We provide two optional arguments for this annotation: An option to clear the taint if the variable is used as the argument to the given sanitization function and an option to mark sink functions that are prohibited from receiving the tainted variable as an argument. The data flow annotation supports tainting various objects: If the object is a variable, the variable itself will be tainted, and if the object is a callable object such as a function, objects returned from the callable object will always be tainted. In example ④, `pwd` is disallowed to reach the `print` function, except if it is passed to `hash` first. By default, the sink is `write()`, which can output data to any untrusted domain (e.g., file system or network).

Object Access can be controlled by two different types of annotations. With `WATCH.[ALLOW|BLOCK](Target Object, Permission)` privilege can be assigned. There are three different types of privileges: A read permission (`r`) grants the program the ability to read the value of an object, including assigning it to other variables or passing it to other modules; an example can be seen in ⑤. Similarly, write permission (`w`) grants the program the ability to modify the value of an object and includes the ability to remove the object.

Lastly, an execute permission (`x`) grants the program the ability to execute an object, providing that the object is a callable like a function or a method, as shown in example 6. The second type of annotation syntax is 7 `WATCH.CON(Target Object)` which means the annotated object's execution time should be statistically constant.

Code Execution annotations can be applied to code regions that should not be allowed to be executed, unless a condition is met. Example 8 forbids execution if `user.type` is not `'admin'`.

Annotation Clearing is used to remove a specific annotation policy or all annotation policies if there are no arguments given. Example 9 disables the specific policy that limits the `execve` system call to the `ls` executable. Example 10 clears policies on all network-related system calls. This type of annotation can be helpful if the user wants to apply different policies for different parts of the code.

B. Policy Monitor

The runtime policy monitor is responsible for translating developer annotations into enforceable runtime policies and identifying the unintended behaviors of the applications as policy violations. To detect such violations, our monitor instruments the program at policy-relevant locations (e.g., object access or system call sites). When an instrumented point is reached during execution, the monitor intercepts the event, evaluates the runtime context against the specified policy, and reports any discrepancy as a security flaw. The following sections detail this instrumentation and enforcement mechanism.

1) Data Flow Monitor: To detect vulnerabilities related to data flow issues like sensitive information exposure, our approach relies on dynamic data flow analysis [43]. A tracking tag (*taint*) is added to the variable annotated by the user, and the data flow monitor will track the data flow and control the taint propagation through the program execution based on the annotation arguments.

As shown in Listing 2, the developer mistakenly outputs the sensitive variable `user_credential` into the local log file. By adding a `Data Flow` annotation (line 2), the user of ANOTA can ask the data flow monitor to track the variable `user_credential` in the program execution and remove the taint when the variable is sanitized. Then the policy violation is detected (line 4) as `user_credential` is being written into a local file, and ANOTA will identify the unintended behavior. Hence, the policy monitor is notified that a new vulnerability has been detected. Writing `hashed_credential`, however, would not trigger the violation because it has been sanitized by the sanitization function `hash` defined in the annotation.

2) System Call Monitor: To identify vulnerabilities associated with unintended system call usage such as path traversal and unrestricted upload of files, the system call monitor module observes the usage, arguments, and return values of system calls. Suppose, for instance, that a user writes annotations to confine read access to specific file paths. To validate whether a `read` system call violates this policy,

```
1 $user_credential = get_credential($user_id);
2 TAINT($user_credential, sanitization=[hash]);
3 $hashed_credential = hash($user_credential);
4 $log_message = sprintf("credential %s
     collected with hash %s\n",
     $user_credential, $hashed_credential);
5 file_put_contents("logfile.txt", $log_message,
      FILE_APPEND);
```

Listing 2. Dataflow example in PHP.

```
def SafeURLOpener(inputLink):
    SYSCALL.NETWORK.BLOCK.SCHEME("file","php"
        ...)
    SYSCALL.NETWORK.BLOCK.HOST("youtube.com"
        ...)
    target = urllib.request.urlopen(inputLink)
    print(target.read())
    SYSCALL.NETWORK.CLEAR()
```

Listing 3. Annotated motivation example.

the path must be extracted from the file descriptor argument of the `read` system call. This argument corresponds to the return value of the corresponding `openat` system call, and the `openat` system call's arguments contain the corresponding file path information. This information is used to determine the file path from which the system call reads data.

At the start of program execution, the system call monitor module attaches itself to the execution process and seamlessly extends its monitoring to subsequent processes spawned by the program. During program execution, the system call monitor inspects system calls to find if any policy violation occurred. If a violation occurs—for instance, an unintended system call is invoked or the program writes data to an unintended location—the system call monitor sends a signal to the execution environment. As a result, the instrumented execution environment could identify that the application is in a vulnerable state.

For the motivating example explained in Section II-B, our approach can successfully detect this vulnerability even if the developer of the code in Listing 1 has no knowledge of the underlying faulty library implementation by adding a few system call annotations, as shown in Listing 3. Note that our annotation system enables developers to transform their intuition about which URLs should and should not be filtered into an explicit and monitored security policy via a small set of annotations. As an additional example, consider Listing 4, where we show an example system call annotation used to detect a path traversal vulnerability. The developer intends to send a file stored in a folder named `static` in the current

```
app.get('/static/:filename', (req, res) => {
    const file = req.params.filename;
    const staticDir = path.join(cwd, 'static');
    SYSCALL.FILE.ALLOW(staticDir);
    res.sendFile(path.join(staticDir, file));}
```

Listing 4. System call example in JavaScript.

```python
if is_auth:
    EXECUTION.BLOCK()
    user_list.append(new_user)
WATCH.ALLOW(user_list, "r")
user_list.remove(existing_user) if is_auth
     else print("not authenticated")
```

Listing 5. Access control example in Python.

working directory to the remote user based on the request URL. An annotation is added in line 4 to ensure that file-related system calls can only access the path specified in the argument.

3) Access Control Monitor: Access control related vulnerabilities, like missing authorization or improper authentication, could be exploited in several ways. For example, executing privileged code logic, reading/writing privileged variables, or interacting with privileged system resources like files and network resources. The developer can mark a section of code as privileged with the `Code Execution` annotation, or a variable as only accessible to the developer with `Object Access` annotations. Inside the instrumented execution environment, ANOTA monitors the code execution status by collecting variable metadata from the stack frame and heap. Then it checks access to variables by watching read and write operations to that variable. Similar to previous policies, the policy monitor will be notified when the policy is violated. As shown in the example in Listing 5, only the administrator should be able to add or remove users.

IV. IMPLEMENTATION

We implement our design as a prototype named ANOTA based on CPython (the original Python interpreter), LLVM [44], and eBPF [45]. The prototype implementation consists of about 5,500 lines of Python, C, and Rust code.

The implementation mirrors the idea of separating policy from mechanism, consisting of two primary components: an Annotation Frontend and a Policy Monitor. The Annotation Frontend is integrated into a modified CPython interpreter to parse annotations and prepare the corresponding security policies. The Policy Monitor enforces these policies and implements the bug sanitizer feedback of the annotations by monitoring program execution.

However, our implementation does not stop at the Python boundary; ANOTA also supports native extension modules written in C/C++. While the Python interpreter cannot inspect the internal operations of this native code, our policy monitor uses different backends depending on the context: (i) For pure Python code, policy enforcement is handled by hooks within our modified CPython interpreter. (ii) For native C/C++ extensions, we implemented an LLVM instrumentation pass to enable annotation support and policy monitoring for these modules. (iii) For system-level interactions (e.g., file system access), we use an eBPF-based [45] monitor to observe system calls made by the application.

Although our prototype implementation targets Python, the underlying design is language-agnostic. Its core components rely on techniques that are generalizable to other ecosystems. For instance, the policy monitor could be adapted for PHP or JavaScript using different instrumentation backends, and the system call module is inherently cross-language.

A. Parsing Annotations to Policy

To integrate our annotations seamlessly into Python without altering the language's syntax, we implemented them to appear as standard function calls. The core of our implementation is an instrumentation hook on the `CALL_FUNCTION` opcode, which handles function invocations. When the `CALL_FUNCTION` opcode is executed, our instrumentation code extracts the function name from the stack and checks if it is one of the annotations. If it is an annotation, the instrumented annotation parsing code will parse additional annotation options from the function arguments into the runtime policies. This is an implementation choice for convenience in Python, not a limitation of our overall design. The core principle of parsing developer-provided syntax to create security policies is language-agnostic. The policy monitor then receives the parsed policy and enforces this policy during subsequent program execution.

B. Vulnerability Detection

Business logic vulnerabilities violate an application's intended semantic policies without causing program crashes so they cannot be detected by traditional sanitizers. We therefore propose a set of custom sanitizers using our annotation system, which enables the explicit definition and identification of these high-level policy violations to detect typical vulnerabilities in our scope.

1) Sensitive Information Leakage: Developers sometimes mistakenly output sensitive information into logs or exception statements, directly or indirectly. To detect such issues, users can use the `Data Flow` class of annotations to mark sensitive data sources (e.g., a variable holding a password) and data sinks (e.g., a logging function). Our system then taints the source variables and tracks the propagation of this taint to track the data flow of those sensitive variables.

The primary implementation challenge is to maintain taint propagation robustly across both pure Python code and native C/C++ extension modules. For the Python code, we modified the CPython interpreter by adding a taint attribute to the base PyObject struct. Since all Python objects inherit from PyObject, this allows any object to carry a taint mark. We then instrumented the CPython to check and propagate this taint attribute for each opcode, effectively tracking data flow during the Python execution [43]. For the cross-language interface, we instrument the argument-parsing functions used by C extension modules, ensuring that taint information is preserved when data moves from Python to C. For the code written in C/C++, we provide a modified version of the LLVM data flow sanitizer. Developers can apply this pass by compiling the extension modules written in C with additional compilation flags and LLVM passes provided by ANOTA to apply the modified data flow sanitizer.

ANOTA can also detect timing side-channel vulnerabilities by statistically testing whether functions that handle sensitive data run in constant time. A classic example in Python is using the "==" to compare the user-provided password against the stored password, the number of matching characters influences the execution time, allowing an attacker to discover the stored password. A function is targeted for this analysis either automatically if it consumes a variable that has been marked sensitive via `Data Flow` and `Object Access` annotation or explicitly if a user uses `WATCH.CON` annotation. ANOTA profile the execution time of functions across different inputs using the algorithm mentioned in Dudect [46], which is selected because of its precision and usability as analyzed by Fourné et al. [47]. If the execution time varies with different inputs, ANOTA will report a potential timing side-channel vulnerability.

2) Improper Access Control: Improper access allows unauthorized users to get access to resources (e.g., admin page or sensitive data) that should only be accessible by privileged users. To detect improper access control vulnerabilities, the `Object Access` and `Code Execution` annotations can be applied to the source code.

`Object Access` annotations entail adding the specified variable to a watch list, along with the privileges defined in the annotation. Instrumentation is then applied to value retrieval and storage opcodes (such as Load_* and Store_*) to monitor operations that involve access or modification of a particular variable. When a variable is added to the watch list, ANOTA initially assesses its scope (global or local). Upon execution of load/store opcodes, ANOTA compares the corresponding variable against those on the watch list. This is done via the `oparg` in Python, which is utilized by the opcode to resolve to the actual argument. Notably, global variables are checked globally, while local variables are only examined within the stack frame where they are defined. Similarly, for Python dynamic modules written in C, ANOTA leverages `ptrace` to set read/write breakpoints to check whether the variable is changed at runtime. When a breakpoint is triggered, a handler function is invoked that sends a signal to the policy monitor system of ANOTA.

In the scenario that the developer knows which part of the code should be accessible to the privileged users, the `Code Execution` annotation is provided to show that the code after the annotations should be privileged-user-only. `Code Execution` annotations are similar to a flipped assertion: If the program execution encounters this annotation and the expression in the argument evaluates to `true`, the program touches a code region that it should not execute.

3) Unintended System Calls: As noted above, vulnerabilities like SSRF, path traversal, and unsafe deserialization all share the common trait of using system calls differently from the developer's intention. To detect these vulnerabilities, developers can use the `System Call` class of annotations to express the intended usages of system calls.

To detect such vulnerabilities, we need to implement a system call monitor module. Using `LD_preload` to hook the system call wrappers in `libc` is not comprehensive, as the modules compiled from C could directly invoke system calls. Therefore, we implement our tool ANOTA on top of Extended Berkeley Packet Filter (eBPF) [45] using the Aya [48] library in around 3,000 lines of Rust code. In detail, the system call monitor leverages eBPF trace points which are a set of reference points or hooks that are attained as the kernel performs a certain task. The eBPF program is attached to two events: `raw_syscalls:sys_enter` and `raw_syscalls:sys_exit`, representing the kernel is about to enter a system call and exit a system call, respectively. The system call ID and arguments are collected in the `raw_syscalls:sys_enter` event, and the return value is collected in the `raw_syscalls:sys_exit` event. Then, this collected system call metadata is added to a hash map.

When the interpreter runs into a `System Call` class of annotations, it parses the annotation to extract the policy. The extracted policy along with the interpreter's process ID is then sent to the system call policy monitor module. The module attaches the eBPF part to the system call trace point and begins to collect data. Every time the hash map is updated, the system call monitor module compares the record against the policy to verify if there is a policy violation. If a policy violation is detected, a signal is sent back to the interpreter and the interpreter triggers a segmentation fault.

ANOTA monitors all file-related system calls, and it is extensible to detect other types of vulnerabilities abusing system calls. One prime example is detecting file-based time-of-check to time-of-use (TOCTOU) vulnerabilities. One typical case of TOCTOU involves a file initially accessed by a system call that checks attributes like `access` or `stat`, followed by another system call that performs actions like writing or reading on the same file. Another type occurs when a program creates a file and, later, changes the privileges after operations, such as writing data to the file. Although this idiom causes a TOCTOU bug, it is still widely used. Thus, to focus on critical reports, we only detect TOCTOU on files containing sensitive information with the help of our data flow analysis. We flag a potential TOCTOU vulnerability when either of these patterns occurs, along with data flow that involves reading or writing sensitive variables to or from the file.

The ability to implement a new detector for TOCTOU by composing our system call monitoring with data flow policies demonstrates the extensibility of the ANOTA framework.

V. EVALUATION

We evaluated our prototype implementation ANOTA in five different sets of experiments which show that ANOTA is capable of identifying various business-logic vulnerabilities listed in the CWE Top 40 in real-world applications, that its runtime overhead is minimal, and that the annotation system is easy to use even for first-time users. We use ANOTA together with the Python fuzzer Atheris [41] which we will refer to as ANOTA+FUZZER. In this setup, during fuzzing with Atheris, ANOTA will promote policy violations to crashes which will be observable by the fuzzer. All experiments were conducted on an Intel Core i9-13900K machine with 64 GB of RAM running Ubuntu 22.04.

In the following, (A) we assess the feasibility of rediscovering known, publicly disclosed business logic vulnerabilities. This study is conducted by an author proficient in both ANOTA and software security to demonstrate that a knowledgeable user can effectively annotate unfamiliar applications, and to confirm that ANOTA's annotations are sufficiently expressive to cover a wide range of vulnerabilities. (B) We apply ANOTA+FUZZER to popular open-source Python-based applications to evaluate if the approach can find new 0-day business-logic vulnerabilities in popular and actively-maintained Python projects. (C) We compare ANOTA+FUZZER against the standard Atheris [41] fuzzer as an ablation study. (D) We perform a usability study to assess the annotation system. We begin with an annotation study recruiting undergraduate and graduate students with varying levels of security and software development expertise to assess the ease of writing annotations for detecting vulnerabilities in unfamiliar applications, establishing a lower bound on effectiveness. Complementing this, we assess the ANOTA's real-world applicability through a study with professional developers and security analysts, focusing on their current detection practices, perceived utility of the tool, and the potential barriers to its adoption in industrial workflows. (E) We measure the runtime performance overhead of ANOTA.

The target applications used in this section include both web applications and standalone Python applications. For standalone Python applications, we use the test cases in the source repository of the respective applications and transform them into a harness for ANOTA+FUZZER. For web applications, we need to use a different strategy, as they receive network requests (instead of getting inputs directly in binary applications). Therefore, we implement a custom mutator for ANOTA+FUZZER to mutate fields in the network request. To make the generated test cases more likely to be valid, the mutator will focus on mutating cookies, query parameters, headers, and URLs. Hence, the fuzzer can generate test cases having an appropriate request format. For applications that need a valid login, ANOTA will keep the session cookie in each request to be able to test the endpoints after the login authentication. To accelerate the fuzzing process, the mutator has a dictionary of serialized malicious payloads and code injection payloads trying to invoke a shell that would be easily captured by the policy monitor mechanism.

A. Rediscovering Known Vulnerabilities

This experiment assesses whether ANOTA can effectively detect various real-world business logic flaws when guided by ANOTA+FUZZER. We begin by curating a list of vulnerable Python packages from the Snyk database [49], filtering for recent, high-impact vulnerabilities that fall within the CWE Top 40. After excluding applications with unsupported dependencies or those that function solely as libraries (such as cryptographic libraries), we established a final set of 47 diverse applications, as listed in Table III. To avoid introducing bias by knowing the detailed vulnerability report, the author who conducts this evaluation only knows the bug type of the application, is only allowed to access the code of the application that should be annotated, and has access to the application's documentation.

TABLE III
KNOWN VULNERABILITIES IN REDISCOVERY EXPERIMENT, WITH ANNOTATION TYPE USED FOR REDISCOVERY AND THE INFORMATION REQUIRED FOR ANNOTATION PER APPLICATION

ID	Name	CWE	Stars $/10^3$	Vuln. Identifier	LoC $/10^3$	Type	Info
1	internetarchive	362	1.5	SNYK-6141253	8	DF	Docs
2	b2-sdk-python	362	0.2	CVE-2022-23651	35	DF	Docs
3	B2_Command_Line_Tool	362	0.5	CVE-2022-23653	15	DF	Docs
4	langchain	918	80.1	CVE-2024-2057	363	SC	Const
5	langchain	918	80.1	CVE-2024-0243	363	SC	Const
6	label-studio	918	16.1	CVE-2023-47116	33	SC	Const
7	whoogle-search	918	8.7	CVE-2024-22205	3	SC	Const
8	gradio	918	27.8	SNYK-6141123	64	SC	Const
9	Paddle	22	21.5	CVE-2024-0818	1056	SC	Const
10	xtts-api-server	22	0.2	SNYK-6398416	3	SC	Const
11	langchain	22	80.1	CVE-2024-28088	363	SC	Const
12	esphome	22	7.4	CVE-2024-27081	371	SC	Const
12	onnx	22	16.6	CVE-2024-27318	121	SC	Const
13	label-studio	434	16.1	SNYK-6347239	33	SC	Const
14	zenml	434	3.6	CVE-2024-28424	187	SC	Const
15	inventree	434	3.6	CVE-2022-2111	99	SC	Const
16	GibsonEnv	502	0.8	CVE-2024-0959	31	SC	Docs
17	Transformers	502	123.0	CVE-2023-6730	1155	SC	Docs
18	synthcity	502	0.3	CVE-2024-0936	46	SC	Docs
19	Apache-Airflow	862	34.0	CVE-2023-50944	658	OA	Docs
20	changedetection.io	306	14.6	CVE-2024-23329	16	CE	Docs
21	aries-cloudagent	287	0.4	CVE-2024-21669	213	OA	Docs
22	MobSF	276	16.1	CVE-2023-42261	23	OA	Docs
23	mlflow	287	17.1	CVE-2023-6014	239	OA	Docs
24	calibre-web	863	11.3	CVE-2022-0405	30	CE	Docs
25	wagtail	200	17.1	CVE-2023-45809	186	DF	API
26	ansible-core	20	60.7	CVE-2024-0690	203	DF	API
27	pyLoad	200	3.1	CVE-2024-21644	52	DF	API
28	omise	200	0.0	SNYK-6138437	8	DF	API
29	airflow-providers-celery	200	34.0	CVE-2023-46215	658	DF	API
30	horizon	601	1.3	CVE-2020-29565	114	SC	Const
31	evennia	601	1.7	SNYK-6591326	151	SC	Const
32	pyLoad	601	3.1	CVE-2024-24808	52	SC	Const
33	llama_index	94	31.7	CVE-2024-3098	305	SC	Docs
34	Aim	94	4.8	CVE-2024-2195	28	SC	Docs
35	vantage6	94	0.0	CVE-2024-21649	35	SC	Docs
36	DIRAC	668	0.1	CVE-2024-29905	225	SC	Docs
37	fonttools	611	4.1	CVE-2023-45139	172	SC	Docs
38	OWSLib	611	0.4	CVE-2023-27476	29	SC	Docs
39	untangle	611	0.6	CVE-2022-31471	1	SC	Docs
40	AccessControl	269	0.0	CVE-2024-51734	9	CE	Docs
41	Jupyter_Core	427	0.2	CVE-2022-39286	3	SC	Docs
42	clearml	522	0.0	CVE-2024-24595	0	DF	Docs
43	indico	639	0.0	CVE-2024-50633	0	OA	Docs
44	vantage6	287	0.0	CVE-2024-21653	35	OA*	—
45	Apache-Superset	287	57.8	CVE-2023-27526	175	OA*	—
46	zenml	287	3.6	CVE-2024-25723	187	OA*	—
47	nautobot-device-onboarding	200	0.0	CVE-2023-48700	3	DF*	—

SC: System Call, DF: Data Flow, OA: Object Access, CE: Code Execution
* **Not rediscovered** due to lack of information

To evaluate an application with ANOTA, we first create a docker container for the application, allowing for easier reproducibility. If we are testing access control vulnerabilities, the fuzzer is only given a low-privilege role, such as a guest user (without registration) instead of a logged-in user or a default user instead of an administrator account. Otherwise, we provide the fuzzer with a privileged account to test the application more comprehensively. Next, we add the required annotations. This procedure took the authors of this work a total of ten working days. Finally, we fuzz each application with ANOTA for 24 hours. If we fail to detect the vulnerability, we check the vulnerability report to analyze the cause of our failure. Table III lists metadata for each application along with the vulnerability identifier and the required annotation. The column *Info* indicates the knowledge used to place the annotation correctly: *Const* indicates knowledge of a constant variable, e.g., confining file accesses to a root directory defined as a constant. *Docs* indicates understanding the context of the application via the documentation. For example, applications with authenticated endpoints should only be accessible to authorized users. Finally, *API* indicates knowledge of third-party API documentation, e.g., that the data returned by an API is sensitive. The asterisk indicates vulnerabilities that could not be reproduced without further knowledge, as described below.

ANOTA+FUZZER can successfully rediscover 43 vulnerabilities. Notably, in the case of `label-studio` (ID 6), not only was the vulnerability detected, but using the same annotation, a bypass for the official patch was also discovered. The patch relies on the remote user to determine the file type, allowing an obfuscated extension name to bypass the check.

ANOTA+FUZZER was unable to detect four vulnerabilities (IDs 44–47 in Table III). The first is in `vantage6` (ID 44), which has an insecure default SSH configuration that allows root access using only password authentication. While password-less authentication is recommended, we did not consider this to be the flagged security issue and, thus, did not add an annotation for it. Otherwise, ANOTA+FUZZER could detect this issue. Similarly, `apache-superset` (ID 45) has a fine-grained privilege control system, which we did not have enough understanding of to create a meaningful annotation. In `zenml` (ID 46), the vulnerability was not triggered due to complex preconditions, where user A needs to create an invitation link and send it to user B. Then, the invited user B will register and craft a request to update user A's password. We verified that ANOTA+FUZZER could theoretically detect this vulnerability by manually providing a seed containing a valid invitation link, showing the importance of the input corpus. In `nautobot-device-onboaring` (ID 47), we added the annotations to the correct place, but the vulnerability will only be triggered under a particular configuration, with which ANOTA+FUZZER also finds the vulnerability.

We successfully reproduce most (43 / 47) of the vulnerabilities only with knowledge of the bug type and project documentation. In the cases where we missed vulnerabilities, maintainers would likely have a higher success rate due to their deeper understanding of the project's business logic.

B. Finding New Vulnerabilities with ANOTA+FUZZER

To evaluate ANOTA+FUZZER's ability to find previously unknown vulnerabilities in real-world applications, we conducted a study on real-world open-source applications collected from GitHub. We selected a set of 60 popular and actively maintained Python projects from GitHub (over 200 stars and recently committed). To test our approach against hardened targets, this set included four projects with active bug bounty programs (Apache Superset, Apache Airflow, ZenML, Calibre-Web) and one that undergoes regular security audits (Home Assistant Core) [50].

TABLE IV
NEW ZERO-DAY VULNERABILITIES FOUND BY ANOTA

ID	Name	CWE	Stars $/10^3$	LoC $/10^3$	Type	Time /min	# Annotations SC	DF	OA	CE
1	OpenAgents	434	3.3	13	SC	60	12	8	11	16
2	ihatemoney	732	1.1	9	OA	30	7	3	10	12
3	Home Assistant core	532	68.0	1629	DF	70	14	9	18	25
4	Apache-Airflow	367	33.9	658	DF	90	19	8	14	19
5	FileCodeBox	532	3.0	5	DF	45	8	4	6	12
6	nebari	532	0.3	14	DF	50	6	5	3	12
7	SolidUI	532	0.5	5	DF	75	13	7	2	13
8	WordOps	532	1.2	14	DF	40	3	8	2	2
9	WordOps	367	1.2	14	DF	40	3	8	2	2
10	ArchiveBox	367	19.3	13	DF	20	6	2	0	0
11	Apache-Superset	434	57.5	175	SC	80	16	4	10	15
12	cmdb	434	1.2	20	SC	40	9	6	8	13
13	zenml	367	3.6	190	DF	65	11	4	7	15
14	MemGPT	208	8.6	25	DF	40	10	6	2	3
15	pyspider	208	16.3	15	DF	55	4	5	8	6
16	alexa_media_player	532	1.3	7	DF	30	2	3	1	0
17	comfyui_controlnet_aux	94	1.4	196	SC	70	10	5	0	2
18	lithops	94	0.3	29	SC	45	13	3	13	3
19	Linly-Talker	94	0.7	37	SC	50	6	7	0	0
20	cmssw	94	1.1	1597	SC	95	20	7	0	2
21	Microsoft RecAI	94	0.4	29	SC	45	5	3	0	0
22	calibre-web	434	11.3	30	SC	85	22	7	13	8

SC: System Call, DF: Data Flow, OA: Object Access, CE: Code Execution
Assigned CVEs: CVE-2024-34524, CVE-2024-37730, CVE-2024-35453, CVE-2024-35454, CVE-2024-35455, CVE-2024-35456, CVE-2024-35457, CVE-2024-35458, CVE-2024-35063, CVE-2024-35064, CVE-2024-35065, CVE-2024-27473, CVE-2024-34524, CVE-2024-34525, CVE-2024-34526, CVE-2024-34527, CVE-2024-34528, CVE-2024-34529

For each project, we simulated a user's workflow with a one-hour time limit to understand the application's core logic and write corresponding annotations. If a project could not be understood within this timeframe, it was skipped. We then fuzzed each annotated application using a harness crafted from its existing test cases. All findings were manually verified.

We reported a total of 22 issues. Of these, 20 were assigned vulnerability identifiers. 17 vulnerabilities are assigned CVE IDs and the remaining 3 confirmed issues received internal tracking identifiers from Apache, GitHub, and Microsoft, respectively (one each). Table IV provides a detailed list of these vulnerabilities. Since an issue is usually found by one individual annotation, we indicate its *Type*. To quantify the required human effort, the table contains the *Time* it took us to create the annotations and the number of each type of annotation we created.

As the table shows, the human effort required was modest; the time to analyze the code and write the necessary annotations was approximately 60 minutes per application, on average. This demonstrates that a user with prior knowledge of ANOTA can become effective on a new and unfamiliar codebase very quickly. In Section V-D1, we discuss more about the usability of ANOTA when the user has no prior knowledge of both ANOTA and the target applications.

Empirically, we found that with moderate human effort, ANOTA can discover 22 previously unknown vulnerabilities (17 CVE assigned), even in hardened targets with active bug bounty programs.

False Positives. We found two false-positive sensitive information leakage vulnerabilities during this experiment. The first false positive is in the `EasyAuth` project. We added a data flow annotation for a variable `token` containing a sensitive authentication token. Indeed, this token is written to a log file, which is detected by ANOTA+FUZZER. However, the token is revoked before logging, making this a false positive. This issue could have been avoided if we had added the token revocation as a taint-removing function. The second false positive occurred in the `Munki` project, which writes the authorization token, to a `curl` configuration file created by `mkstemp()`. We confirmed with the developer that this is the intended behavior and the temporary file is destroyed after use. Both cases are due to our limited understanding of the projects and should be no hindrance for the developers.

C. Ablation Study

To evaluate ANOTA's capabilities, we conducted experiments on a benchmark suite of 35 business logic vulnerabilities (e.g., path traversal, access control flaws) curated from four well-established, intentionally vulnerable Python applications: OWASP's Pygoat [51], OWASP Vulnerable Flask App [52], Damn Small Vulnerable Web (DSVW) [53], and The Vulnerable API (VAmPI) [54]. This suite, comprising $\approx 19,200$ lines of code, provides a ground truth for measuring detection accuracy of our approach.

To evaluate ANOTA's effectiveness as a sanitizer, we conducted an ablation study that isolates its core contribution. We aim to demonstrate that ANOTA provides a standard fuzzer with the necessary sanitizer to detect business logic vulnerabilities that it would otherwise miss. To this end, we compared ANOTA+FUZZER (Atheris + ANOTA) against the baseline fuzzer (Atheris without an additional sanitizer) on our benchmark suite. The results clearly show that our approach significantly enhances the fuzzer's ability to uncover logic flaws: ANOTA+FUZZER successfully detected all 35 vulnerabilities. In contrast, the baseline fuzzer detected none, failing even when directly supplied with the exact inputs known to trigger the flaws. This result demonstrates that a standard fuzzer is blind to these vulnerability classes and that ANOTA provides the essential oracle capability required for their detection.

Next, we also tried to benchmark ANOTA+FUZZER against other established tools to evaluate its performance. A direct comparison with other sanitizers was not feasible, as no tools with ANOTA's similar capability exist. Other fuzzing tools were also unsuitable for a fair comparison due to being overly specialized for single bug types (e.g., EDEFuzz [16], CT-Fuzz [31] SSRFuzz [15]), non-available preconditions (e.g., EDEFuzz [16] for GUI, FLOWFUZZ [21] for perfect resetting or relying on non-portable, heuristic-based methods for different language ecosystems (e.g., Atropos [14] for PHP, ODDFuzz [27] for Java). Therefore, we compared ANOTA+FUZZER against the most relevant state-of-the-art static and dynamic vulnerability scanners. In summary, ANOTA+FUZZER significantly outperforms all scanners in finding the 35 target vulnerabilities. A detailed breakdown of each scanner's performance and evaluation is available in Appendix A.

In summary, ANOTA provides an essential and previously missing sanitizer to standard fuzzers and can also empower fuzzers to outperform existing state-of-the-art scanners by integrating direct developer insight into a dynamic state monitoring framework.

D. User Studies

1) Annotation Study: To evaluate the usability and effectiveness of the annotations in ANOTA, we conducted a user study with 11 voluntarily recruited participants (P1–P11) from two institutions located in different countries. The participants were computer science undergraduate and graduate students with varying levels of security and development expertise, ranging from basic conceptual understanding to hands-on professional experience. None had prior experience with ANOTA or the target applications.

a) Methodology and Preparation: We selected a subset of six applications randomly from Table III and Table IV to cover unique vulnerability types and mitigate the large time investment required for human subject experiments. Each participant was tasked with annotating these six applications. This scenario was designed to be more challenging than a developer annotating their own codebase. Participants were not given internal application knowledge or specific hints about existing vulnerabilities during the trainning period. Instead, they received a high-level summary of the application's purpose, instructions on ANOTA's annotation syntax, and generic examples of code patterns associated with the relevant vulnerability types. We also provide tips like identifying security-critical boundaries where data crosses trust domains and to derive policies from documentation or API specifications. For each task, we measured the time taken, the number of annotations, and the effectiveness of the participants at detecting the known vulnerability.

b) Results: As detailed in Table V, participants demonstrated a high degree of effectiveness despite their lack of familiarity with the target applications. Across 66 total tasks (11 participants × 6 applications), the user-provided annotations successfully detected target vulnerabilities in 55 cases, yielding an 83.3% success rate. The most common failure point (7 of 11) were on a a single, unusually difficult access control vulnerability in `changedetection.io`. Getting this annotation right is difficult, as this flaw required deep, non-local knowledge of the application's intended logic regarding API authorization. Still, four participants' annotations successfully detected this bug. Note that this kind of vulnerability is not an ideal target for ANOTA. This is because if the developer is aware that this endpoint should be protected, they can

TABLE V
USABILITY STUDY RESULT (TIME IS MEASURED IN MINUTES)

	Gradio SSRF				**xtts-api-server** Path Traversal				**cmdb** Unrestricted Upload				**temporai** Untrusted Deserial.				**WordOps** Information Leakage				**changedetection.io** Broken Access Control			
ID	Time	Num	(FP)	Result	Time	Num	(FP)	Result	Time	Num	(FP)	Result	Time	Num	(FP)	Result	Time	Num	(FP)	Result	Time	Num	(FP)	Result
P1	58	20	1	✓	21	6	0	✓	49	3	0	✓	22	5	0	✓	20	17	1	✓	82	23	1	✘
P2	52	10	2	✓	22	8	0	✓	55	12	0	✓	49	5	0	✓	45	19	1	✓	89	19	0	✘
P3	90	76	4	✓	10	10	1	✓	60	46	3	✓	20	3	0	✓	60	40	1	✓	90	68	9	✓
P4	120	16	0	✓	30	10	0	✓	90	10	0	✓	20	5	0	✓	75	16	1	✓	60	8	0	✓
P5	140	69	2	✓	43	41	5	✓	66	127	18	✓	35	15	0	✓	93	90	7	✓	45	64	7	✘
P6	50	36	0	✓	30	13	1	✓	35	72	0	✓	20	3	0	✓	65	62	3	✓	40	16	2	✓
P7	180	36	3	✓	80	15	1	✓	80	76	0	✓	38	5	1	✓	70	38	1	✓	180	93	3	✓
P8	60	54	0	✓	40	20	2	✓	60	123	11	✓	40	24	3	✓	35	62	0	✓	60	69	0	✘
P9	120	33	2	✓	30	3	0	✘	120	61	3	✓	90	11	0	✓	90	21	0	✓	180	69	1	✘
P10	60	13	0	✓	30	21	0	✓	30	23	1	✘	30	1	0	✓	30	14	0	✘	60	13	1	✘
P11	60	22	1	✘	40	13	0	✓	40	17	1	✓	60	2	0	✓	150	29	1	✓	70	5	0	✘
Avg.	86	33			40	15			62	46			45	11			63	28			80	41		

directly fix the issue without the need for testing. Other failures stemmed from overlooking documentation (e.g., P10 missed credential descriptions in `WordOps`) or misunderstanding a specific vulnerability (e.g., P11's misunderstanding of SSRF in `Gradio`).

c) Feedback and Observations: In the post-study feedback, all participants found annotation syntax of ANOTA straightforward to learn and use. They described an initial learning curve, followed by a systematic approach to annotation. Participants adopted a strategy of focusing on one annotation type at a time to reduce context switching. For example, a participant might first identify and annotate sensitive variables (e.g., secrets), then move on to annotate potential system call sites, proceeding sequentially in this manner. Nine participants expressed concerns about potentially overlooking critical locations to annotate, leading them to spend considerable time minimizing this risk during the initial phase. This concern is understandable, as the participants lacked detailed knowledge of the applications under test. However, this issue is unlikely to arise for developers, who are familiar with the code bases they typically work on, even for large projects. Ten participants' feedback mentions they were more confident while writing blocklists compared to allowlists. This outcome was expected, as it is impractical for participants to have comprehensive knowledge of the underlying code. Using blocklists to constrain code behavior proved to be both easier and effective, enabling participants to identify most vulnerabilities in the applications. However, when documentation explicitly defined valid access paths (e.g., constant variables for file paths), participants prefer to use the allowlist since it is more accurate.

d) False Positives: The overall false positive rate was low. After manually inspecting those cases, we find that the following two factors might help reduce the number: the participants tend to write annotations around the code that explicitly express the developer's intention. The participants prefer to use blocklists (over allowlists), which is less likely to cause false positives. False positives are commonly associated with system call annotations within functions containing a few lines of annotated code or data flow tracking across modules. In our study, participants could not iteratively test and refine their policies due to the time and environment constraint. However, in a real-world workflow, developers are familiar with the codebase and can address these issues using an iterative approach, refining annotations as needed to eliminate false positives effectively.

2) Real-world Developer Study: To assess the real-world applicability and adoption potential of ANOTA, we conducted a qualitative study with ten software developers and security analysts, representing a wide range of professional experience. Among the participants, one had 1-3 years of experience, four had 3-5 years, four had 5-10 years, and one had more than 10 years. This study investigated their current practices, their attitude toward adopting ANOTA, and potential barriers to adoption.

The study has three key findings. First, it confirmed the problem's relevance: all (10/10) participants reported that they were aware of and had encountered business-logic vulnerabilities in their projects. When asked about their current detection methods, the responses indicated a heavy reliance on manual Manual Code Review (10/10) and External Penetration Testing (9/10). Other methods included Unit Testing (4/10) and Manual QA Testing (3/10), with only one participant (1/10) having used static analysis for this purpose. This validates that the problem remains a significant, work-intensive challenge. Second, we found universal agreement on ANOTA's value proposition. When asked about the effort-versus-value trade-off, specifically, if spending approximately one hour adding annotations was a reasonable exchange for the ability to detect business-logic vulnerabilities. The response was unanimously positive (10/10). We observed a distinction based on project criticality: four participants stated they would definitely use ANOTA, while the remaining six reported they would apply it to applications based on their importance. This indicates that all of the experienced participants perceive significant value in ANOTA and view the required annotation effort as a reasonable trade-off for the potential security gains. Third, the primary barriers to adoption identified by participants were of a practical nature. The most significant barriers were the learning curve of a new annotation language (10/10) and the effort of integration with CI/CD pipelines (6/10). Additionally, two participants raised concerns that annotations could reduce code readability without a standardized style. While the concern regarding the learning curve is valid, we note that our user study with students (Section V-D1) demonstrated that a brief training session was sufficient for participants to effectively use the ANOTA language. The other concerns regarding integration and standardization are practical implementation challenges that provide clear, valuable directions for future work centered on tooling, usability, and IDE/CI integration.

This feedback validates the perceived value of ANOTA's approach and provides clear, practical directions for future work centered on tooling, usability, and workflow integration.

> Our user studies collectively demonstrate both the practical usability and real-world value of ANOTA. The annotation study demonstrates that effective annotations can be created with minimal prior knowledge of ANOTA and the target application. The real-world study with professional developers showed that the annotation effort required for our approach is considered a reasonable trade-off for its security value.

E. Performance Evaluation

Finally, we evaluate the performance overhead of ANOTA. The runtime performance is measured by using the Python Performance Benchmark Suite [42], which is an authoritative benchmark for alternative Python implementations provided. This fits our purpose, as ANOTA is implemented by modifying the CPython interpreter. All benchmarks are executed in rigorous mode to get accurate data. The taint tracking module is evaluated by tainting every variable, an extreme setup that is not typical for regular usage. Additionally, we measure the syscall module overhead by recording every system call.

> We find that tracing all variables results in an average runtime overhead of 10%, while tracing all system calls leads to an overhead of about 5%.

VI. DISCUSSION AND FUTURE WORK

Our prototype implementation of ANOTA enables developers to express their intuition of the program behavior with annotations. In this section, we differentiate ANOTA from binary analysis frameworks and explore potential future extensions regarding automated annotations and language support.

Our prototype implements a practical engineering approach to cross-language taint analysis, which proved effective in our evaluation. We acknowledge that comprehensive cross-language taint analysis is a challenging research problem. A more robust implementation could build upon existing work, such as PolyCruise [55] for the Python-C interface and other works addressing different language pairs [56], [57].

A. Comparison with DBI Frameworks

As ANOTA is at its core a framework to build custom sanitizers, it might seem similar to Dynamic Binary Instrumentation (DBI) frameworks like Valgrind [7] and DynamoRIO [59] that provide an API to build custom low-level binary analysis tools. However, they represent a *fundamentally* different design paradigm, as summarized in Table VI. There is a strong divergence in the semantic information available, the target users, and the performance characteristics.

TABLE VI
COMPARATIVE ANALYSIS OF ANOTA, VALGRIND, AND DYNAMORIO

Tool	**ANOTA**	**Valgrind**	**DynamoRIO**
Goal	Human-in-the-loop sanitizer framework	Dynamic Binary Instrumentation (DBI) framework for building binary analysis tools	
Target	Source code	Binary	
Interface	Policies defined via intuitive annotations in source code	C API on IR / instructions for building instrumentation tools to manipulate execution	
User	App. developer / Security analyst	Security expert	
Vuln.	Business logic vulnerabilities	Enables detection of any machine-level behavior; pre-built tools cover low-level execution errors (memory safety violations)	
Overhead	*Low:* 10%/5%	*High:* 5,417%*	*High:* 5,513%*

* Lower bound of the performance overhead measured for memory tracing [58]

1) The Semantic Gap: DBI frameworks operate at the machine code layer, observing instructions rather than application objects like variables. They view the execution of a Python program as a stream of instructions from the interpreter binary, remaining blind to the high-level logic those instructions represent. For example, a DBI tool cannot directly identify a `password` object in memory. To implement ANOTA's functionality, a DBI user would need to reverse engineer the CPython interpreter's internal memory layout during runtime. This is an enormous challenge, as structural information is lost in translation (cf. the challenges in implementing binary sanitizers [11]). In contrast, ANOTA operates directly within the CPython interpreter, granting it access to the semantic context from an internal perspective.

2) Target Users: As ANOTA is designed for application developers and security analysts with knowledge of the target application, users can add intuitive annotations to detect vulnerabilities (as shown in Section V-D) without understanding the internal details of ANOTA. In contrast, DBI frameworks require users to also possess expert knowledge of binary analysis, systems programming and the framework's internal APIs to build custom analysis tools.

3) Performance: While it would be theoretically possible to implement ANOTA's functionality using DBI frameworks, their low-level instrumentation comes with high runtime overhead. Memory tracing alone, the first step to implementing something like ANOTA via DBI, introduces a prohibitive overhead of over 5,000% (50x) [58], which is orders of magnitude higher than ANOTA's overhead in our evaluation.

In conclusion, DBI frameworks are ill-suited for detecting business logic vulnerabilities, particularly in high-level languages like Python. ANOTA provides the necessary abstraction and efficiency for sanitizing business logic vulnerabilities, especially in the performance-sensitive context of fuzzing.

B. Target Vulnerabilities

Certain vulnerabilities within the CWE Top 40 are excluded from our prototype because they are can be addressed without the need of annotations. For instance, SQL injection, hard-coded credentials, and Cross-Site Request Forgery (CSRF) can be mitigated immediately via standard practices, such as using prepared statements, removing credentials, or implementing CSRF tokens, once a developer is aware of them. Consequently, annotating these issues would be redundant.

However, more vulnerabilities are often not obvious. ANOTA targets subtle vulnerabilities that are easily overlooked, particularly those buried within complex implementations or opaque third-party libraries. ANOTA's primary utility is to make implicit security intent machine-verifiable. In modern workflows, developers often treat APIs (e.g., `urlparse` in List 1) as black boxes. They may understand a security requirement (e,g., block certain network requests) yet remain unaware of internal library flaws. By annotating this intent, ANOTA detects security vulnerabilities caused by API misuse or underlying bugs. This utility is validated by our discovery of zero-day vulnerabilities in audited projects like Home Assistant, enabling ANOTA to serve as a critical sanitizer where deep dependency introspection is infeasible.

C. Integrating Large Language Models (LLMs)

Annotations require manual effort to analyze the application and the quality of these annotations depends on the user's expertise, which can be a challenge in large codebases or when dealing with unfamiliar libraries. A potential solution is integrating LLMs for semi-automating the generation of these policies. An LLM could reduce manual effort by analyzing source code to propose relevant security annotations and bridge the expertise gap for complex dependencies. However, utilizing LLMs is not a simple solution due to challenges like hallucinations and the required large context windows to process sufficient documentation. These limitations necessitate that any LLM-generated annotations undergo rigorous validation by a human expert to ensure their trustworthiness [60].

Our work establishes the foundational framework to express these policies. We view this framework as the necessary first step to transform the semantic information from developer knowledge into machine-enforceable policies, enabling and supporting future research into reliable automation with LLMs.

D. Porting to Other Languages

While ANOTA can be extended to other languages, such as PHP and JavaScript, to find business logic vulnerabilities, we view this primarily as an engineering challenge, as the fundamental strategy of ANOTA is language-agnostic. We briefly sketch the porting strategy for each specific component of the framework: The *Annotation System* resembling standard function calls could be ported by adding built-in functions or instrumenting the language-specific function invocation mechanism to intercept these calls. Similar to our approach for Python's `CALL_FUNCTION` opcode, a PHP implementation could target `DO_FCALL` or `DO_ICALL`, and JavaScript could target the `Call` opcode. The *System Call Monitor* is implemented using eBPF for Linux kernel hooks, agnostic to the language runtime and therefore directly reusable without modification. The *Data Flow Monitor* could leverage existing dynamic taint tracking solutions such as PHP's Taint extension, Augur [61], taintflow [62] for JavaScript, or libDFT [63] for C/C++. The *Object Access Monitor* requires instrumenting low-level operations for variable access, like PHP's `ZEND_FETCH_R/W/RW` opcodes, or the property load (`Lda*`) and store (`Sta*`) instruction families in JS engines.

VII. RELATED WORK

Security vulnerabilities detection is a vast research area. In this section, we focus on the most relevant prior work in dynamic analysis and code sanitization, particularly concerning non-memory safety and business logic flaws.

Dynamic Sanitizers and System Policies. Established dynamic sanitizers like AddressSanitizer [2], MemorySanitizer [3], LeakSanitizer [64], UBSan [8] and ThreadSanitizer [9] are highly effective for memory corruption and data races but do not target business logic vulnerabilities. System-level policy enforcement mechanisms like Landlock [65] or AppArmor [66] are also distinct, as they enforce coarse-grained, process-wide policies, lacking the fine-grained, runtime-adjustable control needed for specific code blocks or contexts relevant to business logic. Recent sanitizers for non-memory flaws are often limited, targeting narrow vulnerability types (e.g., numerical errors [67], code injections [68]), issues unique to embedded systems [69] or restricted to specific languages like Go [70] or PHP [14]. We also differentiate ANOTA from tools detecting application-specific correctness bugs [71]–[74]. (e.g., broken HTML [73] or misinterpreted SQL [72]), which find functional errors rather than the security-critical policy violations ANOTA targets.

Many fuzzing frameworks integrate custom bug oracles, but these are often limited to specific vulnerability classes (e.g., XSS [12], [19], injection vulnerabilities [12], SSRF [15]) or languages (e.g., Atropos for PHP [14]). Tools targeting information leaks also have different limitations. EDEFuzz [16] can only detects sensitive data exposure in API responses but lacks the business-logic context to determine if data is truly sensitive or check leakage from other channels like log files. Ct-fuzz [31] focuses on low-level side-channel leaks, distinct from ANOTA's whole-system logic analysis. FLOWFUZZ [21] requires complex, deterministic setups and manual instrumentation for data leak detection, and does not cover timing side-channels. Other tools target narrow issues like file uploads [75]–[77] or domain-specific policies like robotics [78]. In short, while prior fuzzing frameworks target specific, predefined bug patterns, ANOTA provides a general framework for defining and detecting violations of application-specific policies.

Finally, several approaches leverage developer input or annotations. IJON [79] uses annotations to guide fuzzers towards deeper application states but still relies on traditional bug oracles (e.g., crashes) rather than detecting new vulnerability classes. ASIDE [80] and Anovul [81] use annotations or code markers to check for access control and authentication flaws, respectively. This concept is similar to ANOTA's `Code Execution` annotation. However, ANOTA distinguishes itself through a significantly broader and more generalizable annotation framework designed to specify and detect a wide spectrum of complex business logic vulnerabilities, extending far beyond access control or state reachability goals.

In summary, while previous work has covered various aspects of business-logic vulnerability detection, ANOTA introduces a novel, annotation-driven dynamic sanitization approach specifically designed to identify security-critical business logic flaws, filling a crucial gap in existing defenses.

VIII. CONCLUSION

In this paper, we present a novel, annotation-based sanitization framework to address the critical challenge of detecting business logic vulnerabilities. Based on our analysis of existing fuzzing sanitizers, we find that they often rely on brittle, automated heuristics that cannot capture the necessary application-specific semantic context. To overcome this, ANOTA empowers developers to directly express an application's intended security policies using a lightweight and intuitive annotation system. By encoding a developer's implicit knowledge into explicit, machine-readable annotations, we open up new classes of vulnerabilities for (semi-)automated bug discovery via dynamic code analysis. To this end, we propose a set of annotations that cover the business logic vulnerabilities in CWE's Top 40 most dangerous software weaknesses.

Our prototype, integrated with a standard fuzzer, called ANOTA+FUZZER, demonstrates the effectiveness of this approach by rediscovering 43 known and detecting 22 previously unknown vulnerabilities in popular, well-maintained open-source projects. A total of 17 CVE identifiers are assigned to our findings at the time of writing. Our annotation study and performance benchmarks further confirmed that the system is easy to use and incurs minimal overhead.

By shifting the focus from inferring behavior with brittle heuristics to enforcing explicitly defined policies, ANOTA represents a step forward in the detection of business logic vulnerabilities. We believe this paradigm of leveraging direct developer insight provides a powerful and extensible foundation for securing the complex applications of today and tomorrow. We hope that our research helps push fuzzing, a proven effective bug-finding technique for low-level programming, into the realm of high-level programming languages.

IX. ETHICS CONSIDERATIONS

Our annotation and real-world developer studies were approved by the authors' institutional Ethical Review Board. With the purpose of the study informed, we obtained consent from all participants, ensuring voluntary participation and the right to opt out. All personally identifiable information was removed prior to analysis. Full study protocols are available in the research artifact. We adhered to coordinated disclosure practices for the 22 vulnerabilities discovered by ANOTA. We reported all issues to the relevant developers according to their security protocols and are actively helping fix them.

ACKNOWLEDGMENT

The project underlying this paper was funded with funds from the Federal Ministry of Transport (BMV) under the funding code 45AVF5A011. The author is responsible for the content of this publication.

REFERENCES

[1] D. Song, J. Lettner, P. Rajasekaran, *et al.*, "Sok: Sanitizing for security," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[2] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX Annual Technical Conference (ATC)*, 2012.

[3] E. Stepanov and K. Serebryany, "MemorySanitizer: Fast detector of uninitialized memory use in C++," in *International Symposium on Code Generation and Optimization (CGO)*, 2015.

[4] J. Chen, W. Diao, Q. Zhao, *et al.*, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[5] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[6] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2011.

[7] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *USENIX Annual Technical Conference (ATC)*, 2005.

[8] The Clang Team. "UndefinedBehaviorSanitizer — Clang documentation." (2013), Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[9] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[10] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[11] J. Schilling, A. Wendler, P. Görz, N. Bars, M. Schloegel, and T. Holz, "A binary-level thread sanitizer or why sanitizing on the binary level is hard," in *USENIX Security Symposium*, Aug. 2024.

[12] E. Trickel, F. Pagani, C. Zhu, *et al.*, "Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[13] J. Zhao, Y. Lu, K. Zhu, Z. Chen, and H. Huang, "Cefuzz: An Directed Fuzzing Framework for PHP RCE Vulnerability," *Electronics*, vol. 11, no. 5, 2022.

[14] E. Güler, S. Schumilo, M. Schloegel, *et al.*, "Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities," in *USENIX Security Symposium*, 2024.

[15] E. Wang, J. Chen, W. Xie, *et al.*, "Where URLs Become Weapons: Automated Discovery of SSRF Vulnerabilities in Web Applications," in *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[16] L. Pan, S. Cohney, T. Murray, and V.-T. Pham, "EDEFuzz: A Web API Fuzzer for Excessive Data Exposures," in *International Conference on Software Engineering (ICSE)*, 2024.

[17] D. Siswanto. "Ppfuzz: A fast tool to scan client-side prototype pollution vulnerability written in Rust." (2022), Available: https://github.com/dwisiswant0/ppfuzz.

[18] J. Huang, J. Zhang, J. Liu, C. Li, and R. Dai, "Ufuzzer: Lightweight detection of php-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2021.

[19] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "WebFuzz: Grey-Box Fuzzing for Web Applications," in *European Symposium on Research in Computer Security (ESORICS)*, 2021.

[20] X. Chen, J. Liu, Y. Zhang, *et al.*, "Webfuzzauto: An automated fuzz testing tool integrating reinforcement learning and large language models for web security," in *International Conference on Information Systems and Computing Technology (ISCTech)*, 2024.

[21] B. Gruner, C.-A. Brust, and A. Zeller, "Finding information leaks with information flow fuzzing," *ACM Trans. Softw. Eng. Methodol.*, Jan. 2025.

[22] MITRE. "Trends in Real-World CWEs: 2019 to 2023." (2023), Available: https://cwe.mitre.org/top25/archive/2023/2023_trends.html#tableView.

[23] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee, "FuzzOrigin: Detecting UXSS vulnerabilities in Browsers through Origin Fuzzing," in *USENIX Security Symposium*, 2022.

[24] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box xss detection," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2014.

[25] S. Neef, L. Kleissner, and J.-P. Seifert, "What all the phuzz is about: A coverage-guided fuzzer for finding vulnerabilities in php web applications," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2024.

[26] Y. Chen, Y. Li, Z. Pan, Y. Lu, J. Chen, and S. Ji, "Uradar: Discovering unrestricted file upload vulnerabilities via adaptive dynamic testing," *IEEE Transactions on Information Forensics and Security*, vol. 19, 2023.

[27] S. Cao, B. He, X. Sun, *et al.*, "Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[28] D. Batham. "OpenRedireX: A fuzzer for detecting open redirect vulnerabilities." (2023), Available: https://github.com/devanshbatham/OpenRedireX.

[29] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection," in *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[30] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[31] S. He, M. Emmi, and G. Ciocarlie, "Ct-fuzz: Fuzzing for timing leaks," in *International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[32] Y. Zhai, Z. Qian, C. Song, *et al.*, "Don't Waste My Efforts: Pruning Redundant Sanitizer Checks by Developer-Implemented Type Checks," in *USENIX Security Symposium*, 2024.

[33] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM (CACM)*, vol. 33, no. 12, 1990.

[34] Google. "ClusterFuzz." (May 2022), Available: https://google.github.io/clusterfuzz/.

[35] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/mechanism separation in hydra," *SIGOPS Oper. Syst. Rev.*, vol. 9, no. 5, Nov. 1975.

[36] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[37] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, *et al.*, "Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[38] X. Wu, L. Xiao, Y. Sun, J. Zhang, T. Ma, and L. He, "A survey of human-in-the-loop for machine learning," *Future Generation Computer Systems*, vol. 135, 2022.

[39] E. Mosqueira-Rey, E. Hernández-Pereira, D. Alonso-Ríos, J. Bobes-Bascarán, and Á. Fernández-Leal, "Human-in-the-loop machine learning: A state of the art," *Artificial Intelligence Review*, vol. 56, no. 4, 2023.

[40] TIOBE Software BV. "TIOBE Index (archived)." (Mar. 2025), Available: https://web.archive.org/web/20250315003022/https://www.tiobe.com/tiobe-index/.

[41] Google. "Atheris: A Coverage-Guided, Native Python Fuzzer." (2023), Available: https://github.com/google/atheris.

[42] The Python Software Foundation. "Pyperformance: Python Performance Benchmark Suite." (2024), Available: https://github.com/python/pyperformance.

[43] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask),” in *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[44] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation,” in *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2004.

[45] eBPF Foundation. “eBPF Core Infrastructure Landscape.” (2024), Available: https://ebpf.io/infrastructure/.

[46] O. Reparaz, J. Balasch, and I. Verbauwhede, “Dude, is my code constant time?” In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[47] M. Fourné, D. D. A. Braga, J. Jancar, *et al.*, ““These results must be false”: A usability evaluation of constant-time analysis tools,” in *USENIX Security Symposium*, 2024.

[48] A. Decina. “Aya: An eBPF library for the Rust programming language.” (2021), Available: https://github.com/aya-rs/aya.

[49] Snyk Limited. “Snyk Vulnerability Database.” (2024), Available: https://security.snyk.io/vuln/pip.

[50] P. Schoutsen and F. Nijhof. “Security audits of Home Assistant.” (2023), Available: https://www.home-assistant.io/blog/2023/10/19/security-audits-of-home-assistant/.

[51] adeyosemanputra. “PyGoat: Intentionally vuln web Application Security in django.” (2023), Available: https://github.com/adeyosemanputra/pygoat.

[52] The OWASP Foundation. “OWASP Vulnerable Flask App.” (2022), Available: https://owasp.org/www-project-vulnerable-flask-app/.

[53] M. Stampar. “Damn small vulnerable web.” (2023), Available: https://github.com/stamparm/DSVW.

[54] Erev0s. “VAmPI: Vulnerable REST API with OWASP top 10 vulnerabilities for security testing.” (2024), Available: https://github.com/erev0s/VAmPI.

[55] W. Li, J. Ming, X. Luo, and H. Cai, “PolyCruise: A Cross-Language dynamic information flow analysis,” in *USENIX Security Symposium*, Boston, MA, Aug. 2022.

[56] S. Kan, Y. Gao, Z. Zhong, and Y. Sui, “Cross-language taint analysis: Generating caller-sensitive native code specification for java,” *IEEE Transactions on Software Engineering*, vol. 50, no. 6, 2024.

[57] J. Kreindl, D. Bonetta, L. Stadler, D. Leopoldseder, and H. Mössenböck, “Multi-language dynamic taint analysis in a polyglot virtual machine,” in *International Conference on Managed Programming Languages and Runtimes (MPLR)*, 2020.

[58] Anota. “ANOTA/Supplementary Materials.” (2025), Available: https://github.com/ANOTA-Sanitizer/ANOTA/tree/main/Supplementary%20Materials.

[59] D. L. Bruening and S. Amarasinghe, “Efficient, transparent, and comprehensive runtime code manipulation,” AAI0807735, Ph.D. dissertation, USA, 2004.

[60] Z. Zhang, C. Wang, Y. Wang, *et al.*, “Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation,” *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025.

[61] M. W. Aldrich, A. Turcotte, M. Blanco, and F. Tip, “Augur: Dynamic taint analysis for asynchronous javascript,” in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[62] A. Khashaev. “TaintFlow: A framework for JavaScript dynamic information flow analysis.” (2016), Available: https://github.com/Invizory/taintflow.

[63] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, “Libdft: Practical dynamic data flow tracking for commodity systems,” in *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2012.

[64] The Clang Team. “LeakSanitizer — Clang 21.0.0git documentation.” (2025), Available: https://clang.llvm.org/docs/LeakSanitizer.html.

[65] M. Salaün. “Landlock: Unprivileged access control – Landlock documentation.” (2024), Available: https://landlock.io/.

[66] AppArmor Developers. “AppArmor.” (2025), Available: https://apparmor.net/.

[67] C. Courbet, “Nsan: A floating-point numerical sanitizer,” in *ACM SIGPLAN International Conference on Compiler Construction (CC)*, 2021.

[68] D. K. Patil and K. Patil, “Automated client-side sanitizer for code injection attacks,” *International Journal of Information Technology and Computer Science*, vol. 8, no. 4, 2016.

[69] J. Liu, Y. Shen, Y. Xu, H. Sun, H. Shi, and Y. Jiang, “Effectively sanitizing embedded operating systems,” in *ACM/IEEE Design Automation Conference (DAC)*, 2024.

[70] C. Wang, H. Sun, Y. Xu, Y. Jiang, H. Zhang, and M. Gu, “Go-sanitizer: Bug-oriented assertion generation for golang,” in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019.

[71] T. Su, Y. Yan, J. Wang, *et al.*, “Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021.

[72] Y. Liang, S. Liu, and H. Hu, “Detecting logical bugs of DBMS with coverage-based guidance,” in *USENIX Security Symposium*, 2022.

[73] S. Artzi, A. Kiezun, J. Dolby, *et al.*, “Finding bugs in web applications using dynamic test generation and explicit-state model checking,” *IEEE Transactions on Software Engineering*, vol. 36, no. 4, 2010.

[74] M. Eshghie, C. Artho, H. Stammler, W. Ahrendt, T. Hildebrandt, and G. Schneider, “HighGuard: Cross-Chain Business Logic Monitoring of Smart Contracts,” in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, Oct. 2024.

[75] J. Huang, Y. Li, J. Zhang, and R. Dai, "Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities," in *Conference on Dependable Systems and Networks (DSN)*, 2019.

[76] T. Lee, S. Wi, S. Lee, and S. Son, "FUSE: Finding File Upload Bugs via Penetration Testing," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[77] J. Huang, J. Zhang, J. Liu, C. Li, and R. Dai, "Ufuzzer: Lightweight detection of php-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2021.

[78] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "Pgfuzz: Policy-guided fuzzing for robotic vehicles," in *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[79] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring Deep State Spaces via Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[80] T. Thomas, B. Chu, H. Lipford, J. Smith, and E. Murphy-Hill, "A study of interactive code annotation for access control vulnerabilities," in *IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC)*, 2015.

[81] M. Ghorbanzadeh and H. Reza Shahriari, "Anovul: Detection of logic vulnerabilities in annotated programs via data and control flow analysis," *IET Information Security*, vol. 14, no. 3, 2020.

[82] S. Lee, S. Wi, and S. Son, "Link: Black-box detection of cross-site scripting vulnerabilities using reinforcement learning," in *International Conference on the World Wide Web (WWW)*, 2022.

[83] H. Liu, S. Chen, R. Feng, *et al.*, "A comprehensive study on quality assurance tools for java," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[84] Semgrep, Inc. "Semgrep: Lightweight static analysis for many languages." (2024), Available: https://github.com/semgrep/semgrep.

[85] SonarSource SA. "Sonarqube: The code quality tool for better code." (2024), Available: https://www.sonarsource.com/products/sonarqube/.

[86] Meta Platforms, Inc. "Pyre: Performant type-checking for python." (2023), Available: https://github.com/facebook/pyre-check.

[87] The ZAP Dev Team. "Zed Attack Proxy (ZAP): The world's most widely used web app scanner." (2024), Available: https://www.zaproxy.org/.

[88] N. Surribas. "Wapiti: The web-application vulnerability scanner." (2023), Available: https://wapiti-scanner.github.io/.

APPENDIX A
COMPARISON WITH STATE-OF-THE-ART SCANNERS

We compare ANOTA+FUZZER against three static and two dynamic analysis tools that share at least three supported vulnerability types with ANOTA+FUZZER and are actively maintained. Note that the selected projects are well-established tools used for comparison in other academic works [12], [14], [82], [83]. We compare against the static tools Semgrep [84], SonarQube [85], and Meta's Pysa [86], as well as the dynamic tools Zed Attack Proxy (ZAP) [87] and Wapiti [88]. All tools were configured according to official documentation, which for tools like Pysa and ZAP involved significant manual effort to define sources, sinks, and access rules. Static tools were provided with source/sink data equivalent to our annotations, though we omitted infeasible project-wide manual configurations (e.g., project-wide type hinting for Pysa) that would far exceed the effort required for ANOTA.

TABLE VII
COMPARATIVE EVALUATION AGAINST SOTA TOOLS

	Pygoat $P = 16$		**FLASK** $P = 5$		**DSVW** $P = 10$		**VAmPI** $P = 4$		**Precision**	**Recall**
	TP	FP	TP	FP	TP	FP	TP	FP		
Semgrep	7	2	3	1	4	0	1	0	83.3%	42.9%
SonarQube	10	0	2	0	6	0	0	0	100.0%	51.4%
Pysa	6	1	3	0	2	0	1	0	92.3%	34.3%
ZAP	5	1	4	0	6	0	2	1	94.1%	48.6%
Wapiti	3	9	2	0	3	1	0	2	40.0%	22.9%
Atheris	0	0	0	0	0	0	0	0	0%	0%
ANOTA+FUZZER	16	0	5	0	10	0	4	0	100.0%	100.0%

TP/FP: True/False Positive, P: Positives. Precision = TP/(TP+FP), Recall = TP/P.

Table VII details the results. ANOTA+FUZZER reports all 35 bugs in scope within the 24-hour fuzzing trial. In contrast, the second-place SonarQube detected only 18 bugs (0 false positives), and ZAP detected 17. These results highlight the core strength of ANOTA: combining dynamic analysis with human-provided semantic context.

As already observed by Güler et al. [14], our findings also suggest that dynamic scanners (ZAP, Wapiti) struggled due to black-box limitations and shallow code coverage. Unlike ANOTA, which can observe internal state, scanners like ZAP often lack the granularity to distinguish between read and write permissions, causing it to miss several vulnerabilities. During the evaluation, we also noticed that correctly configuring the vulnerability detection rules for ZAP requires in-depth knowledge of the tool. We expect an annotation-based approach to be more intuitive for developers.

Static tools (Semgrep, Pysa) suffered from high false negatives due to a lack of runtime information. Pysa's documentation highlights that it demands both source/sink configuration files and extensive manual code annotations for accuracy. While extensive manual type annotation could mitigate issues like "taint collapsing,", annotating full projects impractical. Our approach involved using a thorough configuration file while only annotating code sections pertinent to the vulnerabilities. Pysa also struggles with opaque code (like C extensions), defaulting to assuming taint propagation from arguments to return values. ANOTA only needs the source/sink information and overcomes this limitation by gathering accurate information via dynamic code execution.