Finding Information Leaks with Information Flow Fuzzing

This is the author's version of the work. The definitive Version of Record was published in ACM Transactions on Software Engineering and Methodology (TOSEM), https://dl.acm.org/doi/10.1145/3711902.

BERND GRUNER, German Aerospace Center (DLR), Germany
CLEMENS-ALEXANDER BRUST, German Aerospace Center (DLR), Germany
ANDREAS ZELLER, CISPA Helmholtz Center for Information Security, Germany

We present *information flow fuzzing*, an approach that guides fuzzers towards detecting *information leaks*—information that reaches a third party, but should not. The approach detects information flow by means of *mutations*, checking whether and how mutations to (secret) data affect output and execution:

- First, the fuzzer uses information flow as a *leak oracle*. To this end, for each input, the fuzzer first runs the program regularly. Then, it mutates *secret data* such as a certificate or a password, and re-runs the program giving the original input. If the output changes, the fuzzer has revealed an information leak.
- Second, the fuzzer uses information flow as *guidance*. The fuzzer not only maximizes coverage, but also *changes in coverage* and *changes in data* between the two runs. This increases the likelihood that a mutation will spread to the output.

We have implemented a tool named FLOWFUZZ that *wraps* around a C program under test to provide information-flow based oracles and guidance, allowing for integration with all common fuzzers for C programs. Using a set of subjects representing common information leaks, we investigate (1) whether oracles based on information flow detect information leaks in our subjects; and (2) whether guidance based on information flow improves over standard coverage guidance. All data and tools are available for replication and reproduction.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Search-based software engineering*; • **Security and privacy → Information flow control**.

Additional Key Words and Phrases: fuzzing, information flow

ACM Reference Format:
Bernd Gruner, Clemens-Alexander Brust, and Andreas Zeller. 2025. Finding Information Leaks with Information Flow Fuzzing: This is the author's version of the work. The definitive Version of Record was published in ACM Transactions on Software Engineering and Methodology (TOSEM), https://dl.acm.org/doi/10.1145/3711902.. *ACM Trans. Softw. Eng. Methodol.* x, x, Article x (x 2025), 18 pages. https://doi.org/10.1145/3711902

1 INTRODUCTION

Information leaks—the unintentional release of information to an untrusted environment—are among the most dangerous software vulnerabilities. The 2012 *HeartBleed* vulnerability [11] allowed arbitrary parties to obtain data from the memory of an SSL server, including password and certificate data. *SQL injection* [17] and *script injection* [24] are used by attackers to have the server execute

Authors' addresses: Bernd Gruner, Bernd.Gruner@dlr.de, German Aerospace Center (DLR), Mälzerstraße 3–5, Jena, Germany, 07745; Clemens-Alexander Brust, Clemens-Alexander.Brust@dlr.de, German Aerospace Center (DLR), Mälzerstraße 3–5, Jena, Germany, 07745; Andreas Zeller, zeller@cispa.de, CISPA Helmholtz Center for Information Security, Stuhlsatzenhaus 5, Saarbrücken, Germany, 66123.

1049-331X/2025/x-ARTx
https://doi.org/10.1145/3711902

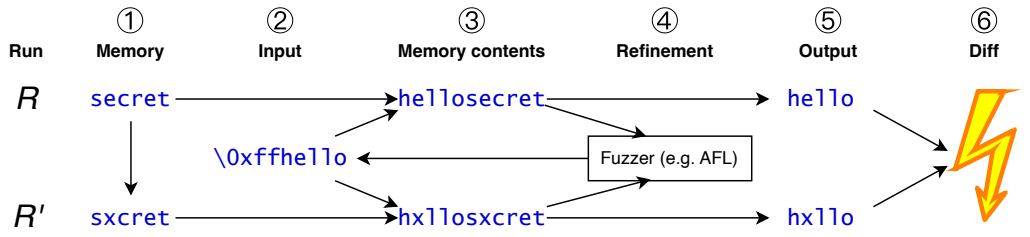

Fig. 1. How FLOWFUZZ works. For each input, FLOWFUZZ runs the program twice, once in the original state, and one with secret data mutated ①. Both runs take the same (system) input from the fuzzer ②. The mutation in the secret propagates throughout program state ③. Differences in state and coverage are passed on to the fuzzer like new coverage ④, effectively maximizing these. If the mutation reaches the output ⑤, FLOWFUZZ detects the difference ⑥ and reports it as a failure.

specific commands, often to exfiltrate data. The 2021 *Log4Shell* vulnerability [22], the "greatest vulnerability ever" allows third parties to execute commands on a server, again used to access sensitive data.

In principle, all such issues can be found through systematic testing. However, fuzzers—software that automatically tests systems using randomly generated data—do not detect information leaks by default. That is because fuzzers only check for *generic* errors such as crashes or hangs. A *sanitizer* that checks for invalid memory accesses or unexpected control flow can extend this range of generic errors. The HeartBleed vulnerability, for instance, can be easily detected using a address sanitizer: The out-of-bounds memory access triggers the sanitizer, which in turn aborts execution. SQL and script injection, however, do not manifest themselves through an illegal memory access or unexpected control flow; and are thus not detected.

Earlier approaches to detect information leaks include data flow analysis [2, 33]. The main problem with both data flow analysis, however, is that it is limited to the program under test—if any data is processed by a third party program, information about its past or future flow will be lost.

In this paper, we follow an *alternate route* to have fuzzers precisely detect information leaks, named *information flow fuzzing*. Rather than analyze or track data flows in code, we check whether *mutations to secret data* impact the program's execution—that is, they change program state, traces, or output. Notably, if the fuzzer can find an input such that the *output* unintentionally *changes* with the data mutation applied, then it has produced proof for an information leak. For instance, when testing for *HeartBleed,* we apply mutations to the stored certificates and passwords. If the server were secure, no client input request should see the response output changed because of the mutation.

We have created a fuzzer named FLOWFUZZ that implements such *mutation-based flow tracking* to detect unexpected information leaks (Figure 1). FLOWFUZZ effectively runs the program *twice* and compares the resulting program states (including outputs) to detect *state differences* induced by the mutation and thus *information flow*. This concept of duplicating functions and variables is an instance of *self-composition,* introduced by Barthe et al. [4]. While self-composition so far has been mostly studied in the context of symbolic verification of small programs, FLOWFUZZ raises self-composition to full-fledged C programs of nontrivial complexity.

FLOWFUZZ uses information flow tracking both as an *oracle* and for *guidance:*

Using information flow tracking as fuzzing oracle. FLOWFUZZ takes a program under test P and a specification of the secret data H. It creates a *wrapper* around P that takes an input I

and then runs P twice: One run R with H unchanged, and one run R' with the same input I, but with H mutated. If the output changes between the two runs, then the wrapper induces a crash. FLOWFUZZ then runs a fuzzer (by default AFL), replacing P with the wrapper created, thus effectively introducing an *oracle* for information leaks.
To the best of our knowledge, this is the first work leveraging mutation-based flow tracking as oracle and guidance for fuzzing software, enabling *common, unchanged fuzzers to detect information leaks and be guided by information flow.*

Using mutation-based flow tracking as fuzzing guidance. Besides leveraging mutation-based flow as an oracle, FLOWFUZZ can also *guide* the fuzzer towards information leaks. To this effect, the wrapper compares the *coverage* as well as the *final memory contents* of R and R': If the coverage has changed, then we have an information flow from the secret towards the functions now executed (or not executed); the more memory contents have changed, the higher the chances of the information propagating to the output.
Our wrapper sends the coverage and data change as an additional *coverage signal* to the fuzzer. This guides the fuzzer to not only maximize coverage, but also to maximize *coverage and data changes* induced by the mutated secret—increasing the chances that these changes will make it to the output as well.
To the best of our knowledge, this is the first work leveraging mutation-based information flow tracking as guidance for fuzzing.

The remainder of this work is organized as follows. Section 2 details the implementation of FLOWFUZZ. Section 3 outlines strategies to control information flows in order to avoid false alarms. Section 4 describes mutation-based information flow tracking using our FLOWFUZZ prototype. Section 5 details how we evaluate our approach, both checking the usefulness of mutation-based information flow as an oracle and as guidance during fuzzing. Section 6 contains the experimental results and answers our research questions. Section 8 relates our approach to other techniques leveraging fuzzing and/or information flow. Section 9 closes with conclusion and future work.

2 THE FLOWFUZZ FUZZER

In the following, we explain the fundamentals of our mutation-based information flow fuzzer, called FLOWFUZZ. First, we address the input specification. Then, we describe the general structure of FLOWFUZZ and how it works. Finally, we detail the implementation of the oracle and guidance.

2.1 Input Specification

FLOWFUZZ requires the following inputs:

Secret Data and Output
Our approach investigates the existence of information flow between some *secret data* and an output. Both the secret data and the output are provided by the user, who can select relevant combinations based on their knowledge of the system. Through a relevant selection, the user can avoid the detection of intentional information flows, described in Section 3. In addition to system-specific secrets, potential examples encompass passwords, certificates, and configurations. Alongside manual inspection, automated procedures can be employed to generate a list of possible outputs [1]. Furthermore, it is possible to mutate multiple secrets and examine multiple outputs simultaneously. This significantly increases efficiency as only one fuzzing campaign needs to be conducted. However, it is no longer uniquely determinable which secret has been leaked, necessitating manual investigation.

Mutation Function
The user is responsible for implementing a *mutation function* that introduces a mutation to the

secret data. The implementation depends on the nature of the secret data. For instance, this might involve modifying a file, overwriting a variable, or altering a configuration. Depending on the kind of secret, the mutation could be a simple bit flip or secret-specific. As a simple example, we have a variable containing a plain text password. Initially, this variable contains an incorrect password, and we use the mutation function to transform the string into another incorrect password, as we do not anticipate any differences in the program's response. The same approach can be applied to certificate files. More detailed examples of such implementations can be found within our experiments.

Fuzzing Harness

As for every fuzzing campaign, a *fuzzing harness* is necessary, providing an entry point into the program under test. The fuzzing harness sets up the necessary program-specific preconditions, receives input from the fuzzer, and forwards it to the program. This harness also integrates the previously mentioned mutation function. Moreover, it utilizes the interface provided by FLOWFUZZ to return the output value. This is an opportunity for users to intercept and manage intentional information flow, thereby preventing FLOWFUZZ from generating an alert.

2.2 Concept and Structure

We have implemented a fuzzer named FLOWFUZZ that uses mutation-based information flow tracking as *oracle* and as *guidance*. Following the sketch given in Section 1, its central characteristics are:

- Take a C program and create a *wrapper* around it, executing the program twice, once *with* and once *without* mutation applied, and if the output differs, crash.
- Be configurable regarding which (secret) data to mutate and how.

To run the program twice, we apply the principle of *self-composition* [4]. We effectively *duplicate* its functions and variables, appending `_prime` to all duplicated identifiers, indicating the run R'. This is done by employing the `'nm'` Unix command to extract pertinent global variables and functions of the program from the symbol table of the object files. Finally, we use the macro `#define` to append `_prime` to the variables and functions.

Listing 1. Pseudocode of FlowFuzz-Wrapper

```
int main(int argc, char *argv[])
{
    main_r(argc, argv);
    mutate(); // apply mutation(s) to secret data
    main_r_prime(argc, argv);
    assert(outputs_are_equal()); // must track this
    compare_globals(); // see below
}
```

The wrapper, containing the implementation of the oracle `assert(outputs_are_equal())` and the guidance `compare_globals()`, is autonomously generated by FLOWFUZZ. It can then be plugged into *any* coverage-guided fuzzer that takes an executable C program; besides AFL++, these can also be grammar-based fuzzers [3, 19] or symbolic fuzzers like KLEE [9].

2.3 Oracle

The oracle compares the output of two runs and triggers an alarm if they differ. The oracle is included within the FLOWFUZZ wrapper, presenting an interface through which the fuzzing harness can transfer the output. It is assumed that the passed value is of type string. The received outputs are compared element by element.

2.4 Guidance

Attaining mutation-based *guidance* requires the transformation of *data differences* into *coverage differences.* We achieve this comparing the values of (duplicated) global variables `var_1_r`, `var_2_r`, and so on. In this context, we exclusively consider global variables that are writable and hence not constant. No further selection of global variables is done, as we assume non-well-behaved programs.

We reuse the extracted global variables from the `nm` command described in Section 2.2 and leverage the Clang library `LibTooling` to get the corresponding data types. Finally, these variables are imported into the FLOWFUZZ wrapper to perform the comparison.

Listing 2. Pseudocode of the function to compare global variables.

```
void compare_globals() {
  int x = 0;
  if (var_1_r != var_1_r_prime)
    x++;
  if (var_2_r != var_2_r_prime)
    x++;
  if (var_3_r != var_3_r_prime)
    x++;
  // and so on for all global vars...
}
```

The whole purpose of this code is to *guide the fuzzer:* Common fuzzers such as AFL++ attempt to increase *edge coverage* in code, including `compare_globals()`, and thus find inputs that execute *as many* `x++` *lines as possible.* This guides the fuzzer to a maximum difference in program state.

C primitive types such as `int`, `long`, and `short` are compared directly. Global variables with type `union` or `struct` are not considered. For pointers and arrays, we compare their *contents,* making each byte difference another coverage target. For each pointer `p` (or array `p` this is the same in C), we proceed as follows: (1) If the size of `p[]` is known at compile time, we assume `p` is an array and compare *all* elements; (2) If `p` is of type `char *`, we assume a `\0`-terminated string and compare all characters; (3) Otherwise, we assume a pointer and compare *one element* (`*p` or `p[0]`) being pointed to. We implement this approximation in order to make FLOWFUZZ as lightweight and fast as possible. A complete comparison would otherwise have to be generated at runtime for each invocation, negatively impacting the performance.

To preserve as many memory contents as possible, FLOWFUZZ provides an option to disable calls to `free()`, keeping heap memory intact (also across the two runs).

With this, the fuzzer favors inputs that *increase differences between global variables,* and thus propagation of mutated data (the "coverage bonus"). Additionally, the fuzzer also favors inputs that cover code only in the code associated with R and R', and thus increase coverage differences, too.

This way, FLOWFUZZ plugs into *any* coverage-guided fuzzer that accepts a C program, such as AFL++ [14].

3 CONTROLLING INFORMATION FLOWS

Executing a program twice can result in *false alarms*—that is, changes in state that are *not* induced by changes to secret data:

- Operating system interaction, notably *time* and *random* operations, trivially change state across invocations. FLOWFUZZ addresses this by recording and resetting time and random state between the two executions.

- *Shared state* in libraries can lead to state changes across invocations. FLOWFUZZ addresses this by filling the stack and respective heap with zero bytes before each execution. This also addresses differences by reading uninitialized memory (which should be treated as a bug and can induce information leaks by itself).
- Finally, *persistent state* across multiple executions can lead to differences in execution. At this point, FLOWFUZZ has no special provision against this issue; users must modify the self-composed code manually to clear up the file system or other parts of the environment. Fortunately, spotting such effects is fairly easy, as persistent state changes typically affect several variables.

All these issues affect other fuzzers and dynamic analyses as well. However, FLOWFUZZ at least allows users to mitigate these issues by altering the generated self-composed code.

In designing the `mutate()` function, users also must distinguish *expected* from *unexpected* information flows. If we mutate the password of some user U, and then find the login procedure now produces an "access denied" output, we have detected an information leak from U's password to U—but this would be expected and tolerable. If we mutate the password of another user V, though, then this should never affect the output to U; if it happened, that would be unexpected.

We emphasize that through careful selection of secret information, output, and `mutation` function, it is possible to reduce intentional information flows. Furthermore, expected information flows can be handled within the fuzzing harness to avoid false alarms. In this paper, we carefully document for each subject how such concerns affect the design of the `mutate()` function, and report the associated effort.

4 INFORMATION FLOW FUZZING: AN EXAMPLE

Let us illustrate information flow fuzzing with FLOWFUZZ on a real-world example. *OpenSSL* is an open-source C library, which provides basic cryptographic and utility functions, including implementations of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. This library is used by applications to provide secure communications including confidentiality, integrity, and authenticity. It is used in the majority of HTTPS web servers as well as for E-Mail transfer, Instant-Messaging and Voice-Over-IP.

The utilization of TLS mandates that the library handles authentication information, including usernames, passwords, and certificates. This information is particularly sensitive, and it must not be accessible to third parties.

TLS has the *Heartbeat* extension [43], which allows the connection between two communication parties to be maintained even if no data is sent for a certain period of time. This process operates through one communicating party sending a heartbeat request, accompanied by a specific payload, to the other party. The reciprocating party responds by transmitting the same payload in return. In the following, we investigate this extension using FLOWFUZZ.

4.1 Setting up FLOWFUZZ

To explore the *Heartbeat* information flow, we make use of FLOWFUZZ together with its default fuzzer, namely AFL++ [14]. AFL++ is a coverage-guided fuzzer for C code that invokes a given function f with a string containing random bytes. We have set up OpenSSL such that FLOWFUZZ can invoke any OpenSSL function f without having to run a server. To do so, we initialize the library, create a context and simulate an incoming request.

For every new input AFL++ produces, it calls the wrapper generated by FLOWFUZZ, as described in Section 2. This wrapper invokes f twice: once with (R'), and once without (R) a mutation applied to the secret. The mutation function is provided by the user (who thus implicitly specifies the secret

data that should not leak out). In our case, we can mutate certificates, passwords, and usernames from the service that uses OpenSSL. Moreover, FLOWFUZZ supports the user by inherently overriding heap and stack memory areas automatically with a constant after the first run. This makes it even easier to detect information leaks caused by memory mishandling.

If the return value of f changes between the two runs R and R', indicating a leak from the heap memory, the wrapper throws an exception, signaling a crash to AFL++. The return value of f is the response's payload in our example. Notice that to detect this information flow, we only need to check the effect of the mutation—neither dynamic nor static information flow analysis is required.

4.2 Using Information Flow as Oracle

Running our FLOWFUZZ prototype on the OpenSSL Heartbeat extension revealed a leak from the heap memory to the response's payload after roughly one second. As seed input, we provided the payload `"heartbeat test"` and the length `"14"`.

The "crash" (the leak) takes place if the length of the actual payload is shorter than the length stated in the request. If the heartbeat request has the parameter `"sent_payload_len = 20"` and `"payload = heartbeat test"` the payload response can appear as follows

- `"heartbeat testaaaaaa"` in R and
- `"heartbeat testbbbbbb"` in R'.

In this response, `"aaaaaa"` and `"bbbbbb"` represent out-of-bounds memory that is likely overwritten by FLOWFUZZ with a constant value, as discussed in Section 3.

How does this leak come to be? In the process of responding to the Heartbeat request, the function `dtls1_process_heartbeat` is responsible for creating the response, including copying the payload from the request into the response. To accomplish this, the payload length provided in the request is utilized. However, this length is not checked, thereby permitting intentionally exaggerated length specifications. These result in not only the copying of the payload from the request but also the incorporation of additional heap memory. An attacker could exploit this vulnerability to access data residing within the heap. Such data might be sensitive information like usernames, passwords, or certificates.

The leak above is known as CVE-2014-0160 [10], was not found by us, and we knew where to search. Still, our experiment demonstrates that **our approach can successfully detect leaks through mutations**—at a low overhead and without any need for static or dynamic data flow analysis.

4.3 Using Information Flow as Guidance

We also have explored if information flow can be used to *guide* the fuzzer towards data flow. To this end, after the execution of each fuzzed function, we extract *all* values of *all* global variables and compare them across the original run R and the mutated run R'.

Again, a difference between variable values would indicate a *flow* from the mutated heap memory towards the response. As discussed earler, we *maximize* such flows using a *coverage bonus*: the `compare_globals()` function sets up FLOWFUZZ that any input that causes such a difference is treated as if it had covered an additional line.

In our OpenSSL example, the HeartBleed vulnerability can be detected at the same pace, with or without the coverage bonus. The secret information is leaked upon being accessed and immediately identified by the oracle within the same run. Therefore, the coverage bonus cannot improve the detection process in this example. The bonus efficacy is thoroughly investigated in subsequent experiments. The bonus is expected to be beneficial to subjects in which the secret information is not leaked immediately upon reading but only under certain conditions.

Table 1. Evaluation Subjects

CVE-ID	Subject	Description
CVE-2021-22898	Curl	Uninitialized data leads to sensitive data exposure
CVE-2017-15924	Shadowsocks-libev	Command injection through improper neutralization
CVE-2020-26298	Redcarpet	Improper neutralization enables injection and CSS
CVE-2021-22876	Curl	Leak from mishandling the HTTP Referer header
CVE-2018-18778	ACME mini_httpd	Path traversal through improper input validation
CVE-2022-29869	Cifs-utils	Mishandling invalid credentials causes leaks
CVE-2014-9938	Git	Remote code execution due to unsanitized variables
CVE-2013-7448	Didiwiki	Path traversal through improper input validation
CVE-2014-0160	OpenSSL	Out of bounds read through improper validation
CVE-2020-5221	Uftpd	Path traversal through improper input validation

5 EVALUATION

To evaluate FLOWFUZZ, we pose the *research questions* already explored in our OpenSSL example (Section 4):

RQ1. Can information flow *oracles* detect common information leaks during fuzzing?
RQ2. How useful is the *guidance* provided by information flow?

Let us first describe the subjects and setup, and then proceed to the actual research questions.

5.1 Evaluation Subjects

We evaluate FLOWFUZZ on a set of *ten C subjects* with varying complexity, which are listed in Table 1.

The subjects are selected from the National Vulnerability Database (NVD). The criteria for selection encompass the requirement that the subject is developed using the C programming language and that its corresponding source code is available. Moreover, each of these has a known CVE concerning a potential information leak and a known input that triggers this very leak.

As we only need simple instrumentation (notably duplicating global variables for guidance), these subjects can be of high complexity, including the use of persistent storage or external programs. For each subject, we set up a dedicated *mutation function* `mutate()` that mutates the secret given in the CVE in a way such that an input exists that leaks the information.

In the Table 2, an overview of the selected secrets and the applied mutations is provided. For CVE-2021-22898 (Curl) and CVE-2014-0160 (OpenSSL), the secret information consists of process memory, which may contain sensitive data. The mutation involves overwriting the heap and stack with a constant value, which FLOWFUZZ does. In the case of CVE-2017-15924 (Shadowsocks-libev) and CVE-2014-9938 (Git), both vulnerabilities enable remote code execution. For simplicity, this example focuses on environment variables; however, it should be noted that remote code execution has the potential to leak significantly more information. During the mutation process, the value of an environment variable is modified.

CVE-2021-22876 (Curl) and CVE-2022-29869 (Cifs-utils) lead to a leakage of passwords. Therefore, passwords are chosen as secret information, with their values altered during mutation process. The vulnerabilities CVE-2018-18778 (ACME mini_httpd) and CVE-2013-7448 (Didiwiki) enable path traversal attacks. The secret information is the content of a file located outside the applications home directory. In a Linux system, this file could be, for example, the password file `passwd`. The mutation involves changing the content of the file.

For CVE-2020-26298 (Redcarpet), which enables injection attacks, the cookie content is used as the secret information, and the mutation modifies the cookie's value. Finally, for CVE-2020-5221 (Uftpd),

Table 2. Summary of Secret Information and Mutation Operations for Each Evaluation Subject

CVE-ID	Secret Information	Mutation
CVE-2021-22898	Content of process memory	Overwritten with constant value
CVE-2017-15924	Environment variables	Change environment variable "user" from "admin" to "user2"
CVE-2020-26298	Cookie content	Change cookie content from "BankSession=123456" to "BankSession=ABCDEFG"
CVE-2021-22876	User Password	Change password from "pass" to "secret"
CVE-2018-18778	Content of file "i"	Change content of file i from "This is secret" to "Another secret"
CVE-2022-29869	User password	Change password from "123456" to "secret"
CVE-2014-9938	Environment variables	Change environment variable "user" from "admin" to "user2"
CVE-2013-7448	Content of file "i"	Change content of file i from "This is secret" to "Another
CVE-2014-0160	Content of process memory	Overwritten with constant value
CVE-2020-5221	Content of the file system	Create a new file

which also enables path traversal, the secret information is the content of the filesystem, and the mutation involves creating a new file in the filesystem.

These selections of secrets and mutations illustrate the broad spectrum of scenarios that FLOWFUZZ can handle. A specific type of mutation is demonstrated in example CVE-2021-22876 (Curl), where the secret is replaced with a string of a different length. Such mutations are applicable in cases where information leaks are not caused by memory management errors.

For each subject, we also determine a set of *seed inputs* that all are typical, yet do not expose the leak. We report the used *seed inputs* for each subject.

Based on our experience, we provide a rough estimate of the manual effort required to apply FLOWFUZZ to a program under test. As outlined in Section 2, a harness is required. Reusing an existing harness, such as one written for use with tools like AFL++ or LIBFUZZER is possible. Expanding such a harness is estimated to take approximately two to four hours. For each subject, it is necessary to identify the secrets, which we estimate will take about one hour. Additionally, implementing a mutation function to modify the secret is estimated to require another hour. Finally, adjusting the build process to integrate FLOWFUZZ with the program under test takes approximately one to two hours.

5.2 Evaluation Setup

We assess FLOWFUZZ in conjunction with the popular AFL++ fuzzer [14]. For each of the subjects listed in Section 5.1, we perform the following. After FLOWFUZZ has created the wrapper, we run AFL++ on the wrapper for 24 hours and assess whether and when its oracle detects the information leak. FLOWFUZZ and AFL++ are run with their default options. For FLOWFUZZ, this means that it provides guidance through data differences inducing coverage bonuses, as shown in Section 2. We repeat all runs ten times to account for randomness during fuzzing, and report averages.

Given the unique nature of our task, involving information leaks that occur only under specific input conditions, it is notable that our research field lacks methods for direct comparison. In the context of fuzzing techniques, the most comparable approach involves the combination of a fuzzer and a sanitizer. To evaluate the effectiveness of our methodology, we conduct comprehensive assessments by executing our experiments with and without the utilization of an *address sanitizer*. This allows us to assess whether the sanitizer can find the information leak.

5.3 RQ1: Usefulness of Oracle

To evaluate RQ1, the usefulness of the oracle, we use the exact setup as described in Section 5.2 and for each subject, report

- how often the FLOWFUZZ oracle (based on mutation leaks) has detected the leak; and
- if the leak is also found via the address sanitizer ASan [40].

Table 3. The table indicates whether the information leak is detected by FLOWFUZZ and the Address Sanitizer (ASan). The first three columns pertain to FLOWFUZZ with different guidance strategies as explained in Section 5.4: (2) standard guidance; (1) coverage differences + standard guidance; full guidance that further enhances the previous strategy by including data differences.

CVE-ID	Subject	Full Guidance	Coverage Diffs + Standard	Standard Guidance	ASan
CVE-2021-22898	Curl	✓	✓	✓	-
CVE-2017-15924	Shadowsocks-libev	✓	✓	✓	-
CVE-2020-26298	Redcarpet	✓	✓	✓	-
CVE-2021-22876	Curl	✓	✓	✓	-
CVE-2018-18778	ACME mini_httpd	✓	✓	✓	-
CVE-2022-29869	Cifs-utils	✓	✓	✓	-
CVE-2014-9938	Git	✓	✓	✓	-
CVE-2013-7448	Didiwiki	✓	✓	✓	-
CVE-2014-0160	OpenSSL	✓	✓	✓	✓
CVE-2020-5221	Uftpd	✓	✓	✓	-

For HeartBleed, for instance, we would expect that the leak would be quickly detected by an address sanitizer (even before the FLOWFUZZ oracle can detect it). For leaks that do not manifest themselves via illegal memory accesses, only the FLOWFUZZ oracles would be effective. We state that leak-based oracles are *effective* if the leak is found in the majority of subjects; notably in cases where address sanitizers are ineffective.

5.4 RQ2: Usefulness of Guidance

To evaluate RQ2, the usefulness of guidance, we perform an *ablation study*. We repeat the experiments from Section 5.3 with

(1) guidance through coverage in R and *coverage differences* between R and R'. To disable guidance through data differences, we comment out the `compare_globals()` functionality.
(2) guidance through coverage in R *only*. This corresponds to the feedback of a standard fuzzing setup. To achieve this, we turn duplication off, and thus have no guidance through data or coverage differences.

For each alternative, we determine

(1) how long and how many invocations it takes to find the leak. We define an invocation as each instance of the fuzzer, in our case AFL++, invoking either the program under test or the FLOWFUZZ wrapper.
(2) the final (code) coverage obtained on the original code

We state that data differences and coverage differences are *effective* if it takes less time or invocations to find leaks, or if the final code coverage is higher. This help to establish the individual benefits of data and coverage differences.

6 EXPERIMENTAL RESULTS

In the following, we answer our two research questions based on the results of our experiments. We assess the oracle in the first research question and evaluate the guidance in the second. The seed inputs used in the experiments, as well as the `mutate()` functions, are included in the replication package.

Table 4. This table presents the average execution time in milliseconds until the information leak is detected by FLOWFUZZ and the address sanitizer (ASan). The first three columns pertain to FLOWFUZZ with different guidance strategies as explained in Section 5.4: (2) standard guidance; (1) coverage differences + standard guidance; and full guidance, which further enhances the previous strategy by including data differences.

CVE-ID	Subject	Full Guidance	Coverage Diffs + Standard	Standard Guidance	ASan
CVE-2021-22898	Curl	9,061,810	**2,040,425**	6,759,551	-
CVE-2017-15924	Shadowsocks-libev	15,845,786	**7,020,257**	8,737,520	-
CVE-2020-26298	Redcarpet	4,486,116	**2,893,487**	9,158,561	-
CVE-2021-22876	Curl	181,101	**17,238**	325,439	-
CVE-2018-18778	ACME mini_httpd	**129,858**	543,757	545,796	-
CVE-2022-29869	Cifs-utils	17	15	**13**	-
CVE-2014-9938	Git	126,484	93,398	**89,174**	-
CVE-2013-7448	Didiwiki	1,677,554	**1,051,109**	2,067,678	-
CVE-2014-0160	OpenSSL	33	35	**18**	49
CVE-2020-5221	Uftpd	1,072	**802**	1,338	-

6.1 RQ1: Usefulness of Oracle

To determine the usefulness of the oracle, we examine ten different subjects containing various types of vulnerabilities that enable information leaks (see Section 5) using FLOWFUZZ and the address sanitizer ASan. Table 3 shows if the information leaks were detected in ten trials. FLOWFUZZ can identify the information leak in all subjects and every trial. In comparison, the address sanitizer can only detect the information leak in the case of the Heartbleed vulnerability in OpenSSL, as it specializes in detecting accesses to memory addresses outside the allocated range. FLOWFUZZ demonstrates its capability to detect information leaks across a wider range of underlying vulnerabilities, such as uninitialized memory, buffer overflow, path traversal, and command injection.

When examining the Heartbleed vulnerability in detail, the address sanitizer required an average of 32 executions to detect the leak, which is fewer than FLOWFUZZ with its full guidance, which required an average of 44 executions. However, FLOWFUZZ, due to its lightweight approach, was still faster on average with 33ms compared to address sanitizers 49ms.

While all ten information leaks could be detected independently of the chosen guidance, there are differences in execution time and the number of invocations. The next research question investigates these differences in detail.

Answer to RQ 1: FLOWFUZZ successfully identified all information leaks in the subjects, thus confirming the usefulness of the oracle, in contrast to the address sanitizer, which only succeeded in one out of ten cases.

6.2 RQ2: Usefulness of Guidance

To address this research question, we conduct an ablation study using three guidance strategies as explained in Section 5.4 (1) standard guidance through edge coverage in R; (2) coverage differences between R and R' + standard guidance; and full guidance, which further enhances the previous strategy by incorporating data differences between R and R'.

Table 4 provides an overview of the average execution time, measured in milliseconds. No strategy consistently performs best across all subjects. However, the coverage differences + standard guidance strategy takes the shortest execution time in six subjects, followed by standard guidance with three subjects and one subject in the case of full guidance.

Table 5. This table presents the average number of invocations until the information leak is detected by FLOWFUZZ and the address sanitizer (ASan). The first three columns pertain to FLOWFUZZ with different guidance strategies as explained in Section 5.4: (2) standard guidance; (1) coverage differences + standard guidance; and full guidance, which further enhances the previous strategy by including data differences.

CVE-ID	Subject	Full Guidance	Coverage Diffs + Standard	Standard Guidance	ASan
CVE-2021-22898	Curl	10,832,314	**2,413,785**	7,924,090	-
CVE-2017-15924	Shadowsocks-libev	13,783,801	**5,881,359**	7,582,584	-
CVE-2020-26298	Redcarpet	17,068	**10,885**	34,859	-
CVE-2021-22876	Curl	49,350	**1,958**	7,575	-
CVE-2018-18778	ACME mini_httpd	**568,085**	2,329,233	2,482,000	-
CVE-2022-29869	Cifs-utils	33	34	**32**	-
CVE-2014-9938	Git	5,046	3,880	**3,565**	-
CVE-2013-7448	Didiwiki	6,557,509	**4,052,282**	8,122,314	-
CVE-2014-0160	OpenSSL	44	36	**32**	**32**
CVE-2020-5221	Uftpd	**353**	453	527	-

The previous results are also reflected in Table 5, which presents the average number of invocations. Here, the coverage differences + standard guidance strategy has the lowest number of invocations for five subjects, followed by standard guidance with three subjects, and full guidance with two subjects.

When comparing coverage illustrated in Table 6, all three methods perform approximately equally, achieving the same coverage on five subjects. In the remaining five subjects, coverage differences + standard guidance and standard guidance each achieve the best coverage in two subjects, and full guidance achieves the best coverage in one subject. It is important to note that our experiments do not aim to optimize coverage but rather focus on detecting information leaks.

In summary, coverage differences + standard guidance strategy demonstrates the best average performance compared to the other strategies. However, the experiments reveal a high standard deviation in execution time and the number of invocations across the ten trials. To conclusively determine the superiority of one strategy over another, a significance test is necessary. This, however, requires more than ten trials per experiment, which exceeds our computational resources, given the extensive number of experiments conducted. Therefore, based on our current experiments, we can recommend the coverage differences + standard guidance strategy. Nevertheless, all guidance strategies successfully detected the information leaks. Furthermore, the choice of the optimal guidance strategy depends significantly on the specifics and structure of the information leak. This includes factors such as the number of global variables used in the project, the extent to which these variables accurately represent the global state of the program, and whether the secret impacts the program's global state.

Answer to RQ 2: In our experiments, the guidance strategy coverage differences + standard guidance performs the best, on average, in terms of execution times and invocations compared to both the standard and the full guidance.

7 THREATS TO VALIDITY

Our evaluation is subject to common threats to validity, which we address in common ways.

The biggest threat, as always, is *external validity,* notably whether the subjects and leaks we selected would be representative of all programs and all leaks. Given that nobody knows all

Table 6. This table presents the average coverage that is achieved when the information leak is detected by FLOWFUZZ and the address sanitizer (ASan). The first three columns pertain to FLOWFUZZ with different guidance strategies as explained in Section 5.4: (2) standard guidance; (1) coverage differences + standard guidance; and full guidance, which further enhances the previous strategy by including data differences.

CVE-ID	Subject	Full Guidance	Coverage Diffs + Standard	Standard Guidance	ASan
CVE-2021-22898	Curl	0.5	0.5	0.5	-
CVE-2017-15924	Shadowsocks-libev	1.2	1.2	1.2	-
CVE-2020-26298	Redcarpet	27.5	27.6	35.9	-
CVE-2021-22876	Curl	2.6	2.6	2.6	-
CVE-2018-18778	ACME mini_httpd	17.4	18.0	17.1	-
CVE-2022-29869	Cifs-utils	4.7	4.8	4.6	-
CVE-2014-9938	Git	64.2	64.2	64.2	-
CVE-2013-7448	Didiwiki	15.7	15.7	15.7	-
CVE-2014-0160	OpenSSL	3.2	3.1	3.0	3.0
CVE-2020-5221	Uftpd	5.3	5.1	5.5	-

programs and all of their leaks, this threat is a concern. We counter this threat by making FLOWFUZZ publicly available, such that any users and researchers can apply it to subjects of their choice.

The second threat to validity is the reliance on *human knowledge*. For each subject, we as experts determined the secret to be preserved, and designed the `mutate()` function accordingly. This expert knowledge influences the performance of FLOWFUZZ. On the other hand, any fuzzer requires some expert knowledge to be set up (such as providing a seed of initial inputs, or determining the means to feed input into the program under test). Furthermore, for detecting unwanted information flow, it is necessary to specify in some form which information flow would be unwanted and which would be tolerated.

Regarding *internal validity,* there is the risk of false alarms from other sources (Section 3), which we assume the user to be able to identify and mitigate. Any such manual steps are documented in Section 6. Having said this, once such issues *are* mitigated, detecting information leaks through mutations would be considered a *gold standard,* as it undoubtedly *shows* the presence of interference (Section 8).

8 RELATED WORK

8.1 Information Flow

The concept of *Information Flow Control,* ensuring that secret information does not leak to third parties, is a pillar of security research. FLOWFUZZ directly implements the principle of *non-interference* [39], distinguishing secret (“high”) and public (“low”) data; a *security policy* would prevent that the (low) input controlled by a third party *interferes* with the (high) secret.

In the past decades, research in the field has mainly focused on *static analysis* and *symbolic verification.* In these domains, showing the *absence* of interference is often too strict to be practical [38], introducing the need for elaborated declassification schemes. In *testing,* however, it suffices to show the *presence* of interference. The MUTAFLOW approach [29] can detect information flow between a source and a sink in cases where static analysis would fail. It employs a mutation of the source and examines the sink for changes. However, this method relies on the existing test cases, limiting its capability to identify only those information flows covered by the given test cases. FLOWFUZZ addresses this limitation by combining mutation-based information flow detection with a fuzzing approach supported by additional guidance.

Using oracles for the detection of information leaks is investigated in a black box differential testing setup by Jung et al. [25]. They change the user information going into the black box system and look for changes in the network data leaving the system. In contrast, FLOWFUZZ is a white-box approach that aims to identify input combinations through fuzzing, potentially leading to the unintended leakage of secret information to the output. Furthermore, information leak oracles are used in the hardware domain, specifically in the context of testing CPU Register-Transfer Level (RTL) designs [23].

Pan et al. [34] investigates Web APIs to identify instances of excessive data exposure. This involves comparing the Document Object Models (DOMs) produced based on simply mutated server-supplied JSON objects.

FLOWFUZZ leverages mutations using *self-composition,* introduced by Barthe et al. [4], to duplicate functions and variables and to compare state. But while self-composition has been mostly studied in symbolic verification of small-scale programs, FLOWFUZZ provides a general framework designed to detect information leakage within software systems and to guide the fuzzing process towards potential leakages.

8.2 Data Flow

Rather than checking information flow through mutations, one can make use of *data flow analysis* [26] to approximate information flow. During analysis, sensitive data is marked with a tag (a *taint*) which is then later passed on to any data that reads the sensitive data. If such a taint reaches the output to the attacker, the information has leaked.

In the field of static taint analysis there exist various methods, including LeakMiner [44], PiOS [12], Amandroid [42] and FlowDroid [2]. The disadvantage of these static approaches is that they cannot exploit runtime information, suffer from overapproximation, and do not scale for real-world projects.

Alternatively, dynamic methods can be leveraged, as presented by Newsome and Song [33] or Enck et al. [13]. However, these methods are subject to potential *overapproximation*—the fact that a secret is *read* or that an output is *written* does not necessarily mean that the secret can also *alter* state or output. This restriction also affects static information flow analysis [7]. Recent efforts combine static analysis with machine learning [21] or dynamic analysis [16] to take advantage of both techniques. Most of the aforementioned approaches focus on the area of smartphone applications and relate to the Android and iOS platforms.

The mentioned analysis methods are based on *data dependencies*: If some instruction *reads* a sensitive variable `H` and *writes* some variable `L`, then `L` inherits the taint from `H`. In testing, data flow is often used as *coverage criterion*—for instance, for each value definition, tests should cover all uses of this value. The *DatAFLow* fuzzer [18] uses lightweight data flow coverage as guidance, detecting some bugs other fuzzers cannot find.

As it comes to *detecting information leaks,* however, data flow analysis has important limitations.

(1) Data flow analysis can produce *false positives,* as some operations read sensitive data and write other data, but do actually not spread information. In `L := H * 0`, as `H` is read and `L` is written, a data flow analysis would assume information flow from `H` to `L`. Assuming absence of nondeterminism (Section 3, mutation-based information flow tracking has only true positives; in the example, a change in the value of `H` will not propagate to `L`.
(2) Data flow analysis does not extend to *data processing outside of the tracked program:* If some data is processed externally (say, stored in a database or processed by an external program), its taint is lost. In contrast, mutations easily survive data processing outside of the tracked program; any change in the result of the external processing indicates an information leak.

On top, *dynamic* data flow analysis has a few additional problems:

(3) Dynamic data flow analysis does not capture *implicit information flow* through control flow, as in `if H then L := true else L := false`. This is not a problem with mutation-based information flow tracking.
(4) Dynamic data flow analysis *slows down* the program under test, as it need to *track* all read and write operations; the DatAFLow authors report a factor of 10× [18]. In contrast, running the program with and without mutated secret data, as with FLOWFUZZ, induces an overhead of only 2× for the execution of R and R' plus a constant expense for the comparison of the global variables.

All these reasons make FLOWFUZZ and information flow fuzzing an important contribution to the state of the art. In future work, however, data flow analysis could prove useful as alternate or additional guidance within tools such as FLOWFUZZ.

8.3 Fuzzing

Fuzz testing or "fuzzing" was introduced by Miller et al. [32] as a technique sending random inputs to programs, testing their robustness. Today's main approaches include:

Evolutionary fuzzing such as *search-based testing* [30, 31] starts with a population of valid *seed inputs* which are then randomly mutated; inputs that get closer towards the testing goal (typically coverage) are preserved and evolved further. The popular AFL [45] and LIBFUZZER [28] tools implement this concept with high efficiency; AFL++ is also the default underlying fuzzer in FLOWFUZZ. Central research directions focus integrating program analysis beyond coverage [37], as well as strategies to direct the fuzzer towards targets of interests [5, 6].

Analysis-based fuzzing makes use of *program code* to determine conditions required to reach specific locations; Seminal representatives include Java Pathfinder [41], KLEE [9], and SAGE [15].

Language-based fuzzing uses language specifications such as *grammars* for producing test inputs [8, 36]; modern implementations include LANGFUZZ [20], PEACH [35], and GRAMMARINATOR [19]. NAUTILUS [3] combines grammar production with code coverage feedback.

The mentioned fuzzing approaches have been adapted and applied in various domains, such as configuration fuzzing [27]. However, they are incapable of identifying information leaks, as their primary focus is solely on detecting crashes and hangs. FLOWFUZZ easily integrates with any of these approaches, as it merely wraps around the program under test and enables the identification of information leaks.

9 CONCLUSION

Fuzzers are great in detecting crashes and hangs, but may miss other serious issues. We present *information flow fuzzing*, a novel approach to have a fuzzer detect *information leaks*. It is superior to static and dynamic data flow analysis because it does not suffer from overapproximation and scalability problems.

Our approach is based on the idea that one can mutate secret data and checking whether and where the mutation propagates; if there is an input that causes the mutation to spread to the output, then we have detected a potential leak. We show that such data mutations can be effective as *oracle* detecting leaks, but also to *guide* the fuzzer towards maximizing the *spread* of such mutations, increasing the chances of information leaks.

Based on our experiments, we demonstrate that FLOWFUZZ can detect all ten information leaks in the subjects with varying underlying vulnerabilities. In contrast, the address sanitizer ASan is only able to identify the information leak in one subject. Furthermore, we found that the guidance strategy coverage differences + standard guidance performs the best on average concerning invocations and execution time in our experiments.

Using a wrapper technique to detect data differences and turning them into coverage differences, our FLOWFUZZ prototype for C programs integrates with all common fuzzers, including AFL++ [14]. Due to extensive automation, FLOWFUZZ can be easily applied to other C programs. All data and tools are available for replication and reproduction.

Our future work will focus on the following topics:

- **Improving the Guidance Strategy:** Since the global state is not equally well represented by global variables across all subjects, it could be beneficial to monitor the corresponding state changes in more detail, for instance, by tracking the input and return values of methods in the program under test.
- **Expand Experiment Scope:** With increased computational resources, the scope of the experiments could be expanded. On the one hand, additional subjects with different characteristics could be included to test FLOWFUZZ's versatility. On the other hand, the existing experiments could be repeated more frequently to perform a significance test, which would allow a definitive determination of whether one guidance strategy is significantly better than the others.

10 DATA AVAILABILITY

The complete replication package for this article is available on GitLab:

- https://gitlab.com/dlr-dw/automated-threat-modeling/flowfuzz

The replication package includes the FLOWFUZZ prototype and, for each subject, a Singularity definition file along with all additional artifacts required to recreate the environment necessary to conduct the experiments described in this paper. For further details regarding the replication package, please refer to the replicated computational results (RCR) report of this work.

ACKNOWLEDGMENTS

This work was done as part of the AVATAR competence cluster, funded by the Federal Ministry of Education and Research (funding code: 16KISA021). With the support of the Helmholtz Information & Data Science Academy (HIDA), financially supported by the HIDA Trainee Network program.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114* (2013).

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[4] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations (CSFW '04)*. IEEE Computer Society, USA, 100.

[5] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. *ACM Transactions on Software Engineering and Methodology* 27, 2, Article 7 (June 2018), 52 pages. https://doi.org/10.1145/3210309

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[7] Dan Boxler and Kristen R Walcott. 2018. Static taint analysis tools to detect information flows. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer …, 46–52.

[8] W. H. Burkhardt. 1967. Generating test programs from syntax. *Computing* 2, 1 (01 Mar 1967), 53–73. https://doi.org/10.1007/BF02235512

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, San Diego, California, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[10] CVE-2014-0160 [n. d.]. CVE-2014-0160. https://nvd.nist.gov/vuln/detail/cve-2014-0160. Retrieved 2023-08-08.

[11] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

[12] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. Pios: Detecting privacy leaks in ios applications.. In *NDSS*. 177–183.

[13] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (jun 2014), 29 pages. https://doi.org/10.1145/2619091

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*.

[15] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[16] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.

[17] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. 2006. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, Vol. 1. IEEE, 13–15.

[18] Adrian Herrera, Mathias Payer, and Antony L. Hosking. 2023. DatAFLow: Toward a Data-Flow-Guided Fuzzer. *ACM Trans. Softw. Eng. Methodol.* (March 2023). https://doi.org/10.1145/3587156

[19] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-Based Open Source Fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/3278186.3278193

[20] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security Symposium*. USENIX Association, Bellevue, WA, 38–38. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[21] Guangwu Hu, Bin Zhang, Xi Xiao, Weizhe Zhang, Long Liao, Ying Zhou, and Xia Yan. 2021. SAMLDroid: A Static Taint Analysis and Machine Learning Combined High-Accuracy Method for Identifying Android Apps with Location Privacy Leakage Risks. *Entropy* 23, 11 (2021). https://doi.org/10.3390/e23111489

[22] Tatum Hunter and Gerrit de Vynck. 2021. The "Most Serious Security Breach Ever" is unfolding right now. *Wall Street Journal* (20 Dec. 2021). https://www.washingtonpost.com/technology/2021/12/20/log4j-hack-vulnerability-java/

[23] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. 2022. SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1473–1487. https://doi.org/10.1145/3548606.3560578

[24] Trevor Jim, Nikhil Swamy, and Michael Hicks. 2007. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*. 601–610.

[25] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '08)*. Association for Computing Machinery, New York, NY, USA, 279–288. https://doi.org/10.1145/1455770.1455806

[26] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2017. *Data flow analysis: theory and practice*. CRC Press.

[27] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. 2024. ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[28] LibFuzzer [n. d.]. LibFuzzer. https://llvm.org/docs/LibFuzzer.html. Retrieved 2022-02-01.

[29] Björn Mathis, Vitalii Avdiienko, Ezekiel O. Soremekun, Marcel Böhme, and Andreas Zeller. 2017. Detecting Information Flow by Mutating Input Data. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 263–273.

[30] Phil McMinn. 2004. Search-Based Software Test Data Generation: A Survey. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156.
[31] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.
[32] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279
[33] James Newsome and Dawn Xiaodong Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.. In *NDSS*, Vol. 5. Citeseer, 3–4.
[34] Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. 2024. EDEFuzz: A Web API Fuzzer for Excessive Data Exposures. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
[35] Peach fuzzer [n. d.]. Peach fuzzer, community edition. https://gitlab.com/peachtech/peach-fuzzer-community. Retrieved 2022-02-12.
[36] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12, 3 (1972), 366–375. https://doi.org/10.1007/BF01932308
[37] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*. http://dx.doi.org/10.14722/ndss.2017.23404
[38] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.
[39] A. Sabelfeld and A. C. Myers. 2006. Language-Based Information-Flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (Sept. 2006), 5—-19. https://doi.org/10.1109/JSAC.2002.806121
[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*. https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker
[41] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (Boston, Massachusetts, USA) *(ISSTA '04)*. Association for Computing Machinery, New York, NY, USA, 97–107. https://doi.org/10.1145/1007512.1007526
[42] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Trans. Priv. Secur.* 21, 3, Article 14 (apr 2018), 32 pages. https://doi.org/10.1145/3183575
[43] Michael Williams, Michael Tüxen, and Robin Seggelmann. 2012. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520. https://doi.org/10.17487/RFC6520
[44] Zhemin Yang and Min Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *2012 Third World Congress on Software Engineering*. 101–104. https://doi.org/10.1109/WCSE.2012.26
[45] Michal Załewski. [n. d.]. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Retrieved 2022-02-01.