Empirical Software Engineering (2025) 30:22
https://doi.org/10.1007/s10664-024-10556-3

Hyperfuzzing: black-box security hypertesting with a grey-box fuzzer

Daniel Blackwell[1] · Ingolf Becker[1] · David Clark[1]

Accepted: 23 September 2024

Abstract
Despite being a severe error where programs inadvertently reveal confidential information, insecure flows rarely receive explicit attention during software testing. LeakFuzzer uses an input-output non-interference property, specialised via a security flow policy for the program under test, to advance the state of the art. It detects insecure flows by using hypertesting for violations of the program's non-interference property. LeakFuzzer extends the capabilities of the state of the art fuzzer, AFL++, and thus inherits its advantages such as scalability, automated input generation, high coverage and low developer intervention. It can thus detect the same set of errors as AFL++, as well as being able to detect violations of secure information flow policies at small additional performance costs. This offers a significant advance in scalability and automation for the state of the art. We evaluated LeakFuzzer on a diverse set of 12 C and C++ benchmarks containing known bugs that cause confidential information to be disclosed, ranging in size from just 80 to over 900k lines of code. Nine of these are taken from real-world CVEs including Heartbleed and a recent error in PostgreSQL. Given 20 24-hour runs, LeakFuzzer can find 100% of the insecure flows in the SUTs whereas existing techniques using the CBMC model checker and AFL++ augmented with different sanitizers can only find 40% at best.

Keywords Information flow control · Information leakage · Fuzzing · Software testing

1 Introduction

In 2015 Johannes Kinder made the case for *automated* hypertesting (Kinder 2015). His motivating example was Heartbleed, the famous violation in OpenSSL of the now standard program flow security hyperproperty (Clarkson and Schneider 2008) called *non-*

Communicated by: Feldt and Zimmermann

Daniel Blackwell
daniel.blackwell.14@ucl.ac.uk

Ingolf Becker
i.becker@ucl.ac.uk

David Clark
david.clark@ucl.ac.uk

1 University College London, London, United Kingdom

Published online: 08 November 2024

interference (Goguen and Meseguer 1982). Until our creation of LeakFuzzer, Kinder's implicit challenge of automatically detecting violations of the non-interference property had not been met. LeakFuzzer is the first tool that automatically generates a set of hypertests and uses them to check that a C/C++ program together with its security policy do not violate the non-interference property. Since LeakFuzzer is a modification of the greybox mutational feedback fuzzer AFL++ (Fioraldi et al. 2020), it not only automatically generates new test inputs but it inherits the fuzzer's exploratory power with respect to branch coverage of the target program.

The key theoretical underpinning for LeakFuzzer is the non-interference property. In Goguen and Meseguer's 1982 paper the property was stated in a general way:

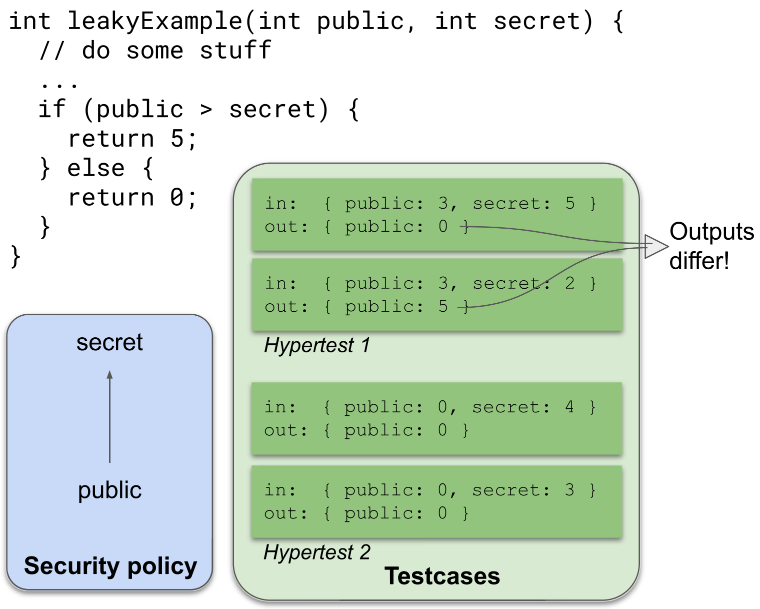
> One group of users, using a certain set of commands, is *noninterfering* with another group of users if what the first group does with those commands has no effect on what the second group of users can see Goguen and Meseguer (1982).

There are many ways this idea can be instantiated and for LeakFuzzer we use a version of this property that frequently occurs in the secure flow literature (Sabelfeld and Myers 2003). We only observe program inputs and outputs to determine whether confidential information has been revealed (hence our characterisation of the security property as *"black box"*). Further, the property only considers terminating inputs and assumes that each input has at most both a high security (*secret*) part and a low security (*non-secret* or *public*) part, similarly for outputs. For example, the interface through which Heartbleed is triggered has a *public* input (the packet sent by the client to the server) and a *public* output (the response sent from the server back to the client), but other parts of internal memory written to by the process during execution must be treated as *secret* as keys or other secrets may be stored there. In this work, we treat any internal memory that is written to—without the express intention of it being output to *public* output—as a *secret* input. This (low security inputs and outputs with internal memory protected as a high input) is a common security policy in practice for multi-user programs / systems (Heusser and Malacaria 2010).

The recognition that non-interference is formally a hyperproperty was published in 2008 by Clarkson and Schneider (2008). They distinguished properties of single program executions, such as non-termination, from security properties such as non-interference or bounds on the amount of confidential information revealed. These latter formal properties must necessarily be expressed in terms of *sets* of executions. As explained in Section 2, our non-interference property is expressed as a universal quantification over *pairs* of program executions. So a failure of the non-interference property must need a pair of tests, i.e. a hypertest, to witness a fault. Figure 1 illustrates the hypertest concept.

In Fig. 1 a function, `leakyExample` is paired with a security policy where `secret` has a higher classification than `public`. The input parameters names align with the security policy names, and the return value of `leakyExample` is visible to public. Hypertest 1 has two inputs, the first has ⟨public = 3, secret = 5⟩, the second has ⟨public = 3, secret = 2⟩. Note that only the value of `secret` changes between the two inputs. However the (public) output values from the function `leakyExample` are different. The forgoing is a single hypertest (pair of tests in this case) that exposes a violation of the non-interference property by the program/policy pair. Not every hypertest exposes a violation as shown by hypertest 2 in Fig. 1. Security properties are further discussed in Section 2.

The Heartbleed bug in the OpenSSL cryptography library resulted in a buffer over-read, which exposed secret program memory information to attackers. Buffer over-reads are automatically detectable by fuzzing in combination with MemorySanitizer, now included with Clang and GCC. However, most buffer over-read bugs result in program crashes, it is rare



```
int leakyExample(int public, int secret) {
  // do some stuff
  ...
  if (public > secret) {
    return 5;
  } else {
    return 0;
  }
}
```

```
in:  { public: 3, secret: 5 }
out: { public: 0 }

in:  { public: 3, secret: 2 }
out: { public: 5 }
```

```
in:  { public: 0, secret: 4 }
out: { public: 0 }

in:  { public: 0, secret: 3 }
out: { public: 0 }
```

Fig. 1 Synthetic code example with an accompanying security policy and two hypertests, one exposing a violation of non-interference and the other failing to do so

for them to cause confidential information to be revealed to an unprivileged user. As a result, when these bugs are detected with MemorySanitizer they are usually assigned a lower priority, because the tool has no knowledge of the information leakage. LeakFuzzer is capable of detecting the insecure flow, indicating the severity of the bug. The strength of hypertesting against a security property is both that insecure flow detection is completely general, i.e. independent of the nature of the flow; and that there are no uncertainties. Either a hypertest witnesses a violation or it does not.

Existing benchmarks only include a small number of real programs (Heusser and Malacaria 2010), or a large number of toy programs (Hamann et al. 2018), which violate secure flow properties. To better evaluate the extended capabilities of LeakFuzzer, we collected a set of 12 C/C++ programs of varying sizes, from 82 to 905,264 lines of code, nine of which contain insecure flows that have been assigned CVE numbers, into a test bench we call Secure Information Flow Faults (SIFF). We have used these SUTs to compare LeakFuzzer against existing state of the art approaches to insecure flow detection. Specifically we have compared LeakFuzzer against a model checking approach, that uses the C Bounded Model Checker (CBMC), and conventional AFL++ augmented in turn with two different sanitisers, MemorySanitizer (MSan) and DataFlowSanitizer (DFSan). Unlike CBMC, the mutational engine of AFL++ is nondeterministic so multiple runs were required for comparison purposes. Each setup was run 20 times for up to 24-hours (the same as FuzzBench Metzman et al. 2021), over 400 CPU days of cumulative evaluation was performed.

LeakFuzzer found every insecure flow in every program but not with every campaign, however the majority of the 24 hour LeakFuzzer campaigns did find the sought insecure flow. In comparison to LeakFuzzer, AFL++, augmented with either sanitizer, was able to detect insecure flows in 5/12 SIFF SUTs while CBMC could detect insecure flows in 3/12 SIFF SUTs. LeakFuzzer tended to use more memory than AFL++ and less than CBMC, which exhausted the available 128GB of memory on one SUT, but there were no issues with memory exhaustion for LeakFuzzer in any experiment.

The contributions of this paper can be summarised as:

- The creation of LeakFuzzer, the first hyperfuzzer to test against an input-output non-interference property,
- The creation of the Secure Information Flow Faults benchmark suite (SIFF), a set of varied C/C++ programs for assessing insecure flow detection tools,
- A rigorous evaluation study using SIFF that assesses the efficacy and efficiency of LeakFuzzer and compares it with three other tools: CBMC, and AFL++ augmented first with DataFlowSanitizer then with MemorySanitizer.

2 Background

In this section we elaborate the concepts on which previous insecure flow detection tools and LeakFuzzer are built: security policies, the non-interference property, hypertesting and greybox fuzzing.

2.1 The Information Flow Control Problem

Information outside programs (inputs and outputs) that travels over networks can be encrypted, providing a level of security. The interface(s) to programs may need to distinguish between different groups of users based on their privileged access to information, hence the access control problem. But implicit within the problem of designing programs with correct access control for different privilege groups is that of maintaining correct information flow control during program operation. A program that correctly implements encryption and access control may still be vulnerable if its operation inadvertently leaks information between privilege groups. It is this last problem that LeakFuzzer addresses. All three techniques, encryption, access control, and flow security are required for "end to end" security of computing with information.

Contemporary approaches to information flow control use a policy for the program, variously called a *secure flow policy*, a *non-interference policy*, or simply a *security policy*, depending on the context. One of the earliest and best known, developed for the USA military, is the 1973 Bell Lapadula model of access control that provided flow guarantees if the model was followed (Bell and LaPadula 1973). Their work formalised security policies and introduced the notion of a security classification for each piece of information in a program. Denning subsequently recognised that the Bell and LaPadula principle of "no read ups and no write downs" between higher privilege and lower privilege groups produced an ordering, in fact a partial order, and that this could be extended to a lattice ordering, for completeness (Denning 1976). She then introduced Lattice Based Access Control (LBAC) in which program data containers are mapped to lattice points that represent access control categories. The lattice ordering is interpreted as that information can only flow within the program from a lower classification to a higher one. To sum up, program users are divided into privilege groups as are data containers (variables etc.) within the program. A lattice is constructed with nodes labelled with the privilege groups and the ordering of the lattice expresses the constraints on information flow between them. The simplest and most widely used lattice for security policies is the two point or *High-Low* lattice with the ordering $Low \sqsubseteq High$. In this paper every program discussed or experimented on has a security policy that is a mapping of its data containers to either *High* or *Low* in the *High*-*Low* lattice.

A program and a security policy being correctly aligned is expressed via a security property. The commonly used security property for many program-policy pairs stems from the 1982 paper by Goguen and Meseguer (1982) which introduced the *non-interference principle* described in Section 1. This is commonly interpreted as an input-output property in terms of the behaviour of the program and this is what LeakFuzzer uses:

> A program *P* satisfies non-interference if and only if for any pair of low equivalent initial states, the resulting final states from running *P* with these initial states are also low equivalent.

We assume each state of the program can be partitioned into two parts, a *High* and a *Low* part. Low equivalence of a pair of states means that the parts of the two states that are labelled as *Low* contain the same values. Essentially, LeakFuzzer is using the non-interference property as a (hyper)test oracle.

As another example, consider a simple program that only takes input from a *High* user, and makes all output available to any user (i.e. to *Low* users):

```
password = read_from_high()
print(password)
```

As usual our secure flow policy is based on the *High*-*Low* lattice. The only data container in the program is the variable `password` that is labelled *High* as it is written by input from a high user. Since the data in `password` is printed, it is effectively made available to *Low* users. From the source code, it is clear that the *secret* value `password` is copied to the program output visible to *Low*; and thus violates the security policy. We can witness this violation by providing a pair of inputs that differ only in their *High* part, and consequently differ in their *Low* output. For this program any pair of inputs are *Low* equivalent as there is no *Low* labelled data container, hence a non-interference violation witnessing hypertest could be the pair of *High* inputs "test" and "pass" that produce *Low* outputs "test" and "pass". The inputs were *Low* equivalent and the (low) outputs differed.

2.2 Side-Channel Leakage

At a high level of abstraction, side-channel leakage is leakage of information through means other than program output; for example through execution time or power consumption. Being able to detect differences in these observables that are dependent on the *High* input values allows an attacker to deduce information about those values. Consider the following program:

```
1 guess = read_from_low()
2 password = read_from_high()
3 if guess.length != password.length:
4     exit
5 for i in 0..guess.length:
6     if guess[i] != password[i]:
7         exit
```

Here the security policy again uses the *High*-*Low* lattice and `read_from_low()` takes input from *Low*, and `read_from_high()` takes input from *High*, thus the variables `guess` and `password` contain *Low* and *High* information respectively. Firstly the length of the input is checked against the length of the actual password, exiting if they are not equal. The ‘for’ loop in lines 5-7 checks each character of the input `guess` against the actual password, and when any single character does not match the password, it exits. Assuming that *Low* is able to observe the runtime of this process, and `read_from_high()` always takes the same amount of time, *Low* can learn information about the degree of correctness of their guess. The longer the program runtime, the closer their guess is to being correct and thus the more information they learn about the value of `password`.

Notably there is no output from this program. Outputting the result (correct or incorrect) would in itself leak information about the value of `password`.

2.3 Self-Composition

Self-composition (Barthe et al. 2011), as the name suggests, involves composing a program with itself in such a way as to provide the two executions with the same *Low* inputs but differing *High* (secret) inputs; the *Low* outputs can then be compared to check that they match (indicating non-interference). This approach is used by side-channel fuzzers (discussed in Section 3.1), as well as the model checking approach that we compare against in the evaluation of LeakFuzzer. Self-composition is useful as it can convert a hypertest into a single test. See the following example where the input parameter `public` is classified as *Low* and `secret` is *High*:

```
int isLarge(int public, int secret) {
    // do some stuff here, does not matter what
    ...
    if (secret > 2) return 1; else return 0;
}
```

To implement the self-composition approach we would create the following wrapper function that compares the output of `isLarge` with two different `secret` values:

```
int selfComposedIsLarge(int pub, int sec1, int sec2) {
    assert(isLarge(pub, sec1) == isLarge(pub, sec2));
}
```

Any testcase for the wrapper function that causes the assertion to fail here, for example ⟨ `pub = 0, sec1 = 1, sec2 = 3` ⟩, is a witness to the insecure flow of secret information in `isLarge`. Essentially self-composition allows a hypertest to be performed with a single testcase (given a wrapper function), thus allowing it to be used by techniques that do not natively support hypertesting.

2.4 Fuzzing

Fuzzing is program testing technique. Generally it is considered a system testing technique, but could be applied to smaller units, and is generally applied with the intention of discovering security vulnerabilities. The term fuzzing dates back to at least 1990 (Miller et al. 1990). At its core, it is closely related to random testing and the process is orchestrated by a tool called a fuzzer. The fuzzer is responsible for generating test cases, executing them and watching for crashes within the program. The term *fuzzing campaign* is used to refer to the fuzzing process from start to finish. These fuzzing campaigns run for a long time, regularly at least 24 hours (Klees et al. 2018), but often much longer (Moroz 2019).

Like software testing in general, there are 2 opposing top-level approaches to building a fuzzer: a black-box approach which has no knowledge of the SUT's (System Under Test) internal structure and state, whereas a white-box approach uses program analysis to improve exploration of program behaviours. An example black-box fuzzer is Zzuf (Hocevar 2007), the key advantage of the black-box approach is that test cases can be generated with virtually no overhead and fuzzing speed only depends on the program runtime. An example white-box fuzzer is SAGE (Godefroid et al. 2012) which leverages symbolic execution to allow it to systematically test different execution paths; this process suffers from the typical limitations of symbolic execution, which limits scalability. Indeed some of the approaches based around fuzzing are incredibly complicated, take for example DriFuzz (Shen et al. 2024), which uses concolic execution to figure out how to successfully initialise hardware device drivers; these drivers are executed inside of a full system emulation (using QEMU Bellard 2005).

There is one further class of fuzzer, lying between black-box and white-box classes is the aptly named grey-box class, one of the most well known is American Fuzzy Lop (Zalewski 2014) (AFL) and its updated relative AFL++ (Fioraldi et al. 2020). AFL instruments the binary for the SUT in such a way as to receive coverage information detailing which branches were run in an execution; testcases are mutated and mutations run, with the fuzzer storing any that lead to new coverage in a queue for further mutation. Using this queue-based mutation approach, AFL is able to explore a wide range of the SUT's behaviours without the program size and complexity constraints of a white-box fuzzer. AFL includes a compiler pass for LLVM that generates and inserts the necessary instrumentation (but can compile only C/C++), but it is also capable of fuzzing uninstrumented binaries using its built in QEMU mode. There are many extensions to AFL to support Python, Rust, JavaScript, Java, Swift, OCaml and .net (Google 2019).

2.5 Sanitizers

Whilst fuzzing primarily aims to detect crashes in programs, modifications can be made to the program under test to allow for a broader range of errors to be detected. One of the simplest ways to achieve this is to add assert statements that trigger a crash whenever a bad state is reached.

There is a family of these error detectors included in the LLVM compiler infrastructure referred to as *sanitizers* (Google 2011). As such, they are compiled into the program under test and doing this is usually as simple as adding a compilation flag for C and C++ programs. When an error is detected by the sanitizer at runtime, the program is crashed and thus the error detected by the fuzzer. The sanitizer dumps useful information such as stack trace and other details before exiting, which aids in the debugging process later on.

AddressSanitizer detects addressability related memory issues such as use after free, buffer overflows and use after scope; it also includes LeakSanitizer which detects memory leaks. ThreadSanitizer detects data races and deadlocks, MemorySanitizer detects use of uninitialised memory and UndefinedBehaviourSanitizer detects undefined behaviour such as integer overflow and bitwise shifts out of bounds. DataFlowSanitizer is an implementation of dynamic taint analysis, which requires modifying the test program to insert API calls to label data and check for label propagation.

3 Related Work

3.1 Fuzzing Applied to Side-Channel Leakage

Fuzzing has been applied to side-channel leakage detection, firstly in 2019 by a tool called DifFuzz (Nilizadeh et al. 2019). DifFuzz works on Java programs, and makes use of self-composition. It executes the target program with two differing secret inputs, and counts the number of JVM (Java virtual machine) bytecode instructions executed in each case; if these differ then it is flagged as a timing leak. A follow on paper of DifFuzz is QFuzz (Noller and Tizpaz-Niari 2021), which also provides an estimate on the quantity of confidential information leaked through the timing side-channel.

A 2020 paper describes a fuzzing tool—ct-fuzz (He et al. 2020)—for detecting side-channel leakage in C and C++ programs. Like DifFuzz, it uses self-composition, but has a more complex model for estimating runtime: firstly it checks that both executions follow the exact same path, and additionally a CPU cache model is used to determine whether any differences in cache misses exist.

Another 2020 paper describes a fuzzing-based approach for detecting JIT-induced side-channels in the JVM (Brennan et al. 2020). Here JIT refers to just-in-time compilation, which is used to compile heavily used sections of code at runtime, speeding them up. Statistical methods are used to determine the amount of secret information that can be learned from the differences in execution time caused by the JIT compilation.

Note that the first three papers (Nilizadeh et al. 2019; Noller and Tizpaz-Niari 2021; He et al. 2020) use models to *estimate* runtime differences, which can vary significantly between hardware setups. The final paper (Brennan et al. 2020) overcomes this by using real execution times combined with statistics, but would ideally still need each program to be tested on each possible target hardware setup. In contrast to LeakFuzzer, none of these tools are able to detect insecure information flows to program output, as the detection of side-channel leakage is a fundamentally different problem.

3.2 Quantified Information Flow

There is a body of work on static or dynamic analysis for measuring the size of insecure flows that began with Clark et al. (2007). Detecting an insecure flow and measuring one are two different things, but depending on the algorithm for measuring it, they can overlap. Heusser's work falls into this category (Heusser and Malacaria 2010) as does that of Biondi mentioned above (Biondi et al. 2018) as well as the tool LeakiEst (Chothia et al. 2013). These approaches either use static analysis and do not scale to larger programs (like CBMC discussed below), or are dynamic but do not automatically generate inputs, like LeakiEst or LeakWatch (Chothia et al. 2014).

3.3 Constraint Solving Approaches to Quantifying Information Flow

An alternative approach to LeakFuzzer is constraint based model checking, a formal verification technique. In 2010, Heusser and Malacaria (2010) used the CBMC (Bounded Model Checker for C) tool to provide a lower bound on the quantity of information leaked through insecure flows in C programs, including evaluation on six real world programs (four of which contained reported CVEs). The method described there iterates over guesses at the channel capacity of an insecure flow so that an upper bound on it can eventually be found.

Integral to the process is that it discovers by constraint solving whether two or more distinct outputs can be produced as a counter example to a channel capacity assertion by providing the program with inputs that do not differ in their non-secret part.

A 2012 paper by Phan et al. (2012) used symbolic execution to quantify information flow, in this case using Java Pathfinder (JPF), which as the name suggests operates on Java bytecode programs. This technique is similar to that explained above, but was evaluated only on fabricated example programs.

It is acknowledged that there are issues with scaling these verification techniques to work with larger programs. A proposed solution is to approximate the model to reduce solving complexity (Biondi et al. 2018), although so far this has only been implemented in a simplified toy programming language.

3.4 Hypertesting

Recent papers on HyperGI, a technique that detect insecure flows and repairs them also use hypertests (Mesecan et al. 2021, 2022). However, they do not have a method for automatic hypertest generation.

4 LeakFuzzer

Formal verification techniques for discovering insecure information flows within programs tend to suffer from an inability to scale, unlike software validation (testing) techniques. However many validation-based approaches do not solve the problem of testcase generation, which is a significant issue for programs accepting complex inputs. In comparison, fuzzing is a validation technique that is noted for its ability to handle testcase generation, achieve good program exploration and scale well to large programs. Here we present LeakFuzzer, a fuzzing-based tool developed with the goal of providing automated, scalable insecure flow detection.

LeakFuzzer is based on the popular AFL++ fuzzer (Fioraldi et al. 2020), and as such contains all of the basic components of a standard grey-box fuzzer as used for finding regular program errors; these are shown in black in Fig. 2. There is an *input queue* that stores 'interesting' inputs; in LeakFuzzer's case this is unmodified from AFL++, whereby an interesting input is one that previously achieved unseen program coverage and therefore should be mutated further to generate new inputs by the *mutation engine*. The *forkserver* creates instances of the system under test (SUT), feeds them the mutated inputs and collects coverage information. Finally, the *decision engine* receives the coverage information and decides whether this input is 'interesting' and thus should be stored in the *input queue*.

We have made a number of architectural modifications and additions in order to detect a new class of error: insecure flows. The conceptual changes are shown in green in Fig. 2. Firstly, program output is fed back through the forkserver to the decision engine. Secondly the decision engine has been modified to process program output and store input pairs that expose insecure flows. Finally, the mutation engine has been modified as described in section 4.2.

LeakFuzzer is able to detect a superset of the errors that AFL++ can. The hypertesting approach used in LeakFuzzer can be generalised to test for any hyperproperty, and we call a fuzzer that implements such an approach a hyperfuzzer. Note that in the following sections a simple security policy is assumed, with two classifications *secret* and *public*, where *secret* is of a higher classification than *public*.

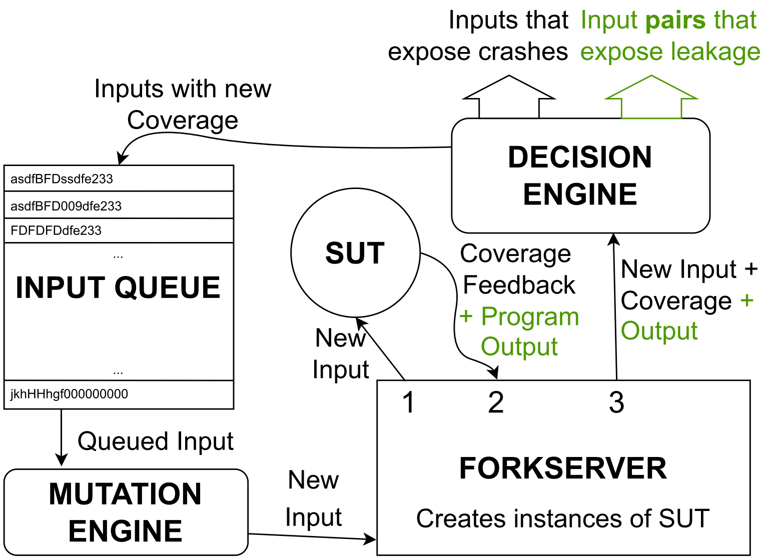


Fig. 2 *High*-level architectural overview of a grey-box fuzzer (black), with additions for LeakFuzzer (green)

4.1 Hypertesting Approach

By storing a map from program inputs to outputs LeakFuzzer only needs to run each input once. This contrasts with the efficiency of methods that use self-composition (discussed in section 2.3) to achieve hypertests. Suppose that there are N distinct *High* labelled values in the test set. For each possible *Low* labelled input, self composition must in the worst case explore a space of C_2^N explicit pairs, i.e. $O(N^2)$, as opposed to LeakFuzzer's exploration of a space of at most $O(N)$.

In order to illustrate this, let's consider this simple program which takes no public inputs, one secret input—a 2-bit unsigned integer `int` (taking values between 0 and 3 inclusive)—and returns a value that is visible to public:

```
int isLarge(int secret) {
  if (secret > 2) return 1; else return 0;
}
```

Using the self-composition approach we could create the following wrapper function:

```
int selfComposedIsLarge(int secret1, int secret2) {
  assert(isLarge(secret1) == isLarge(secret2));
}
```

The six possible non-repeating inputs to `selfComposedIsLarge` are `{0, 1}`, `{0, 2}`, `{0, 3}`, `{1, 2}`, `{1, 3}`, `{2, 3}`. Of these, `{0, 3}`, `{1, 3}` and `{2, 3}` cause the assertion to fail. This means that three out of the six ($\frac{1}{2}$) hypertests expose the insecure flow from *secret*. Therefore if we were to sample hypertest pairs uniformly, we would expect that two ($1 \div \frac{1}{2}$) hypertests need evaluation before we detect the flow. This means running `isLarge` four times; but it could potentially be as high as eight times in the worst case (that is, the three hypertest pairs that do not detect the flow are chosen first).

The approach taken by LeakFuzzer instead requires each input to be tested no more than once by caching the output so that it can be compared against in future without needing to be rerun. Caching outputs in full would require too much memory, hence we store 64-bit hashes of program outputs alongside 64-bit hashes of the corresponding program public and secret inputs. Hashes are capable of colliding, however for a 64-bit hash, collisions are unlikely to effect a fuzzing campaign with 1 billion ($\approx 2^{30}$) or so inputs significantly. The XXH64 hash (4973 2014) is used due to it's extremely high throughput. Not only can we store and compare outputs for programs with large output spaces, but the comparisons can be done in a single machine code instruction due to the size of the stored hashes. This sacrifices some memory efficiency compared to naive self-composition—which requires no memory of past inputs and outputs—in exchange for improved time efficiency as each input can be executed just once.

A pseudocode listing of the insecure flow detection algorithm used by LeakFuzzer can be found in Fig. 3, and line numbers in the following prose reference this figure. A hash table is held in memory by LeakFuzzer, with the public input hashes used as keys, that map to values consisting of a structure containing secret input hashes, output hashes and other properties (lines 6–11). Using this hash table we can detect differences in output within pairs of testcases where public inputs are identical, and secret inputs differ. This is sufficient to infer that an insecure flow does occur.

This does not ensure that those output differences are not caused by some form of non-determinism as opposed to being deterministically influenced by the secret input. To mitigate this, LeakFuzzer stores any testcases that have resulted in a different output to other testcases with an identical public input, reruns them 100 times and directly compares the output buffers byte-by-byte, with any testcases that give flaky (inconsistent) outputs being discarded (lines 26–31). Any testcases that pass this *flakiness test* are stored in full in the hash table value's structure (line 36), and once a pair of non-flaky, output differing testcases has been found for any public input hash then both are reported as an insecure hypertest pair and output to file (lines 38–42).

4.2 Generating {Public, Secret} Input Pairs

The basis for LeakFuzzer, AFL++, has no awareness of the internal structure of inputs by default, and as such generates only raw byte arrays. It is up to the *fuzzing harness* to parse these byte arrays into a format that can be used by the program being tested. This is, at its core, a similar process to deserialisation.

A naive approach to deserialising a fuzzer generated input into public and secret parts could be taken if either the public or secret parts were of fixed length x. One could simply read in the first $min(\text{input.length},\ x)$ bytes needed to fill the fixed length part, and then read the remaining $max(\text{input.length} - x,\ 0)$ bytes into the other part. If both parts are of fixed length x and y bytes, then the $min(\text{input.length} - x - y,\ 0)$ bytes beyond those required can simply be discarded.

In the case that both input parts can be of variable length, then the fuzzer generated input could be split in half, with the first part used as input for *Low* and the latter used as input for *High*. Alternatively, the first 1 (or more) byte(s) could be interpreted as an integer x indicating the length of the secret part of the input; the first x bytes after the length indicator would be read into the secret input, and the remaining bytes (if any) read into the public input.

These simple approaches require no adaptations to the fuzzer, however the fuzzer has no awareness of which parts of the generated input correspond to the public and secret inputs.

```
struct Input {
 ByteArray public
 ByteArray secret
};

struct HashValue {
 int64 secretInputHashes[]
 int64 publicOutputHashes[]

 ByteArray secretInputsFull[]
};

HashTable<int64, HashValue> dict

# Keep going indefinitely until fuzzing campaign halted
while true:
   Input generatedInput = generateNewInput()
   ByteArray publicOutput = targetProgram.run(generatedInput)

   int64 publicInputHash = hash(generatedInput.public)
   int64 secretInputHash = hash(generatedInput.secret)
   int64 publicOutputHash = hash(publicOutput)

   if (HashValue value = dict.get(publicInputHash)) != null:
      if not value.publicOutputHashes.contains(publicOutputHash):
         is_flaky = false
         for _ in 0..100:
            if targetProgram.run(generatedInput) != publicOutput:
               is_flaky = true
               break
         if not is_flaky:
            value.secretInputHashes.append(secretInputHash)
            value.publicOutputHashes.append(publicOutputHash)

            # Only store the full input if a suspected leak is found
            value.secretInputsFull.append(generatedInput.secret)

            cnt = secretInputsFull.length
            if cnt % 2 == 0:
               outputHypertestToFile(generatedInput.public,
                                 value.secretInputsFull[cnt-2],
                                 value.secretInputsFull[cnt-1])

   else: # publicInputHash not found in dict
      HashValue newValue = {
         secretInputHashes = [secretInputHash],
         publicOutputHashes = [publicOutputHash],
         secretInputsFull = []
      }
      dict.set(publicInputHash, newValue)
```

Fig. 3 Pseudocode algorithm describing the hypertesting approach used by LeakFuzzer

In order to detect differences in output caused by insecure flows, inputs must only differ in their secret parts. Without awareness of which parts of the input are secret, the fuzzer cannot be certain of whether any observed differences are due to the public or secret parts of the generated inputs.

In order to overcome this difficulty, LeakFuzzer is adapted to have oversight of public and secret parts of generated inputs. This separation of input segments has several advantages. First and foremost, it allows LeakFuzzer to determine whether output differences are caused by insecure flows (assuming a fully deterministic program). Secondly, it allows for specific targeting of mutations on either part of the input (as selected from the fuzzer *input queue*).

We have discussed why a hypertest exposing an insecure flow will have identical public inputs, but differing secret inputs, so we may wish to schedule a period that mutates the secret part of the input exclusively. Figure 4 shows the memory structure of the stored inputs, a visualisation of the mutation phases and finally the encoded input exactly as it will be provided to the SUT (note that fuzzing harnesses must decode this back into byte arrays, and a utility function to do this is included with the LeakFuzzer source).

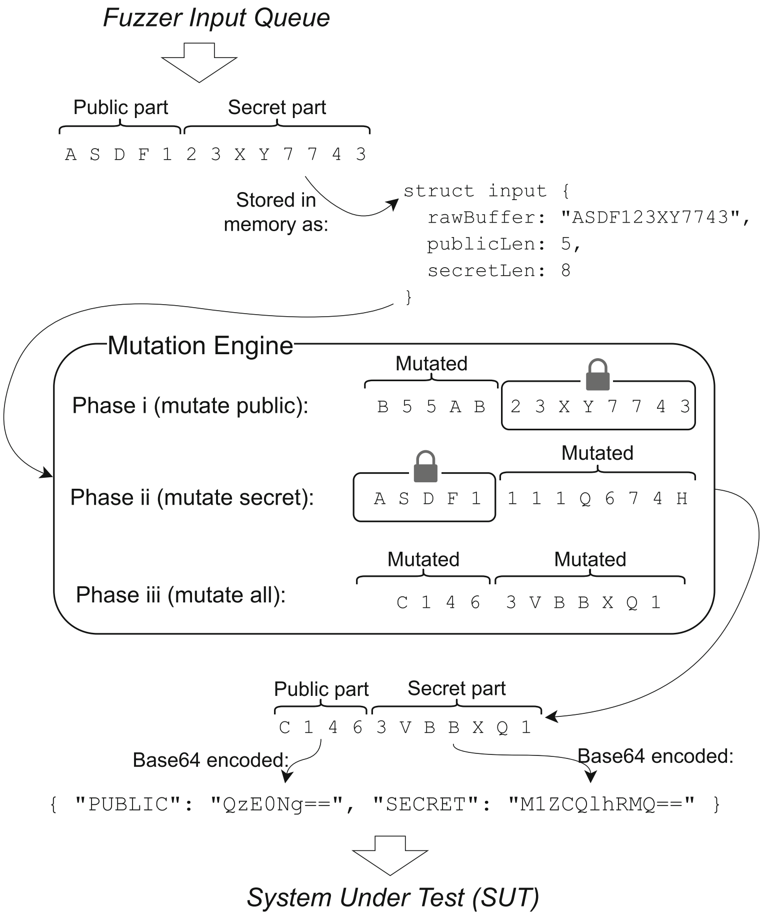


Fig. 4 Block diagram showing the internal and external representations of inputs in LeakFuzzer

LeakFuzzer has three mutation phases targeting: the public part of the input, the secret part of the input, and the entire input. All three strategies are used because any part of the program input could decide whether a particular path is chosen, and branch conditions may depend on some aspect of both the public *and* secret parts of the input. Note that the lengths of both the public and secret parts can be altered by the mutations, and this is accounted for in the fuzzer logic.

The set of available mutations that can be applied to each part of the input are unaltered from AFL++, these are:

- bitflips: flip 1, 2 or 4 consecutive input bits
- byte-flips: flipping all bits in 1, 2 or 4 consecutive input bytes
- arithmetic mutations: adding or subtracting 8-, 16-, or 32-bit values from a set of consecutive input bytes
- *interesting* value substitutions: setting 1, 2, 4 or 8 consecutive input bytes to *interesting* values such as 0, UNSIGNED_MAX, SIGNED_MAX, 1, -1 etc.
- dictionary substitutions: replacing input bytes with constant values that were collected at compile time (for example from conditional expressions) – this is effective for breaking through ‘magic value’ checks
- cloning: copy a random slice of the input and insert it somewhere in the input
- deletion: delete a random slice of the input
- splicing: select another input from the *input queue*, take a random slice of it; then either replace part of the current input with this slice, or insert it somewhere in the current input.

These mutations are applied in a random order, as per the default for AFL++. Initially only one mutation is applied to an input in order to generate a new input for execution. Once a fuzzing campaign has fuzzed every queued input at least once, and has failed to discover any new coverage for at least 60 minutes, then the number of applied mutations before execution is increased by one – allowing the fuzzer to generate more diverse inputs, in the hope of busting out of the current exploration plateau.

4.3 Handling Invalid Memory Reads

The following subsections detail the types of memory misusage that can lead to insecure flows, and how LeakFuzzer can detect this.

4.3.1 Uninitialised Memory Reads

LeakFuzzer targets C and C++ programs, both of which require manual memory management. One of the consequences of this is that variables are not automatically initialised at declaration. This uninitialised variable (section of memory) then contains the values that previously occupied this section of memory; this information potentially contains secrets. One would assume that this type of error is rare, however when using `structs` this mistake is much easier to make. It is common to declare a struct instance, and then populate the member values manually. This approach does avoid writing to each member twice (first to zero it, then secondly to set it to the final value), however it also means that if the developer forgets to set a member then this is left uninitialised. If this uninitialised value is then output from the program, it can reveal secret information.

4.3.2 Out of Bounds Memory Reads

Closely related to uninitialised memory usage is out of bounds memory usage, as both can cause insecure flows due to the reading of values that were not assigned to in the current scope. Heartbleed was caused by data from out of bounds buffer access being copied to program output. An additional source of these errors that can exist in C and C++ is the use of `memcpy` with structs. This is due to *memory alignment*, which is done for performance reasons. Take, for example, the following struct:

```
struct instruction_t {
  char direction;
  int distance
};
```

A tightly packed structure on a 64-bit machine (with 32-bit `ints` as per gcc) would take up only 5 bytes: 1 byte for `direction` followed by 4 bytes for `distance`. However, assuming a preference for 4-byte memory alignment, this struct would actually take up 8 bytes: 1 byte for `direction`, 3 bytes of padding to reach a 4-byte boundary, and 4 bytes for `distance`. Naively serialising this struct to program output by using `memcpy`, the 3 bytes of uninitialised padding would also be output, potentially revealing secret information.

4.3.3 Solution

The *fuzzing harness* that is used to convert the raw byte-string generated by the fuzzer into a form of input accepted by the program often runs only a single input through a slice of the system. As memory pages are zeroed by the OS before being passed to the program process, it is common that early uninitialised memory reads access these zeroed regions. The longer that the program runs, the more common it is that memory is reused; and this is particularly the case in long running interactive programs such as web servers and databases. It is therefore possible that invalid memory reads that lead to confidential information being output can go unnoticed in these fuzzing harnesses, but would affect live long-running processes. Additionally, the output would change depending on the previous value stored in that area of memory and thus may appear to exhibit a form of non-determinism.

In order to allow for the detection of these potential insecure flows, LeakFuzzer uses a technique to set internal program memory to a pattern of consistent non-zero values. To do this, a portion of the generated *secret* input is used as a seed for a pseudorandom number generator, and a sequence of bytes is generated using this (in the current implementation, 8 bytes). This sequence of bytes is then replicated, until it fills the process stack memory, by a function within the initialisation phase of the fuzzing harness. Heap memory initialisation is handled by a provided wrapper for the `malloc` function, this wrapper first allocates the required memory plus an additional 8-bytes and then fills it with the repeated pseudorandom byte sequence. The additional 8-bytes cause differing outputs in the case of buffer overreads.

By seeding the pseudorandom sequence from the *secret* input, it becomes possible for LeakFuzzer to generate hypertests that can produce differing *deterministic outputs* due to invalid memory reads and thus expose the insecure flow.

4.3.4 Worked Example: Heartbleed

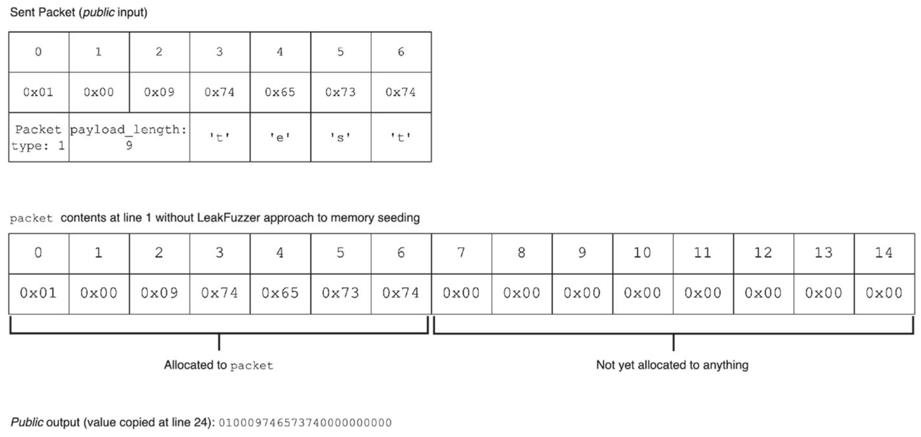
Below is an abbreviated slice of the source code containing the Heartbleed bug:

```
 1  char *packet = ...; // array of bytes sent by the 'public' level user
 2  unsigned int payload_length;
 3  ...
 4
 5  // Read from the payload_length field from the request packet
 6  n2s(packet, payload_length);
 7  ...
 8
 9  // buffer stores the response packet
10  unsigned char *buffer, *bp;
11
12  // Allocate memory for the response, size is 1 byte
13  // message type, plus 2 bytes payload_length, plus
14  // payload_length, plus 16 bytes padding. Note that
15  // this buffer is never accessed out-of-bounds, instead
16  // it is 'packet' that gets overread
17  output_buffer = OPENSSL_malloc(1 + 2 + payload_length + 16);
18
19  ...
20
21  // copy 'payload_length' bytes from 'pl' to 'output_buffer' where:
22  // 'pl' is a pointer to the payload within 'packet'
23  // and 'output_buffer' is the buffer that will be returned to the user
24  memcpy(output_buffer + 3, pl, payload_length);
25
26  ...
27  OPENSSL_free(buffer);
28  // send the populated buffer back to the 'public' level user
29  r = ssl3_write_bytes(
30          s, TLS1_RT_HEARTBEAT, output_buffer,
31          3 + payload + padding
32      );
```

The issue here is that on line 6, the value of `payload_length` is read from the *public input* `packet`; this number of bytes is then copied to the *public* program output `buffer` on line 24. The *public* user is able to set `payload_length` to a value between 0 and 65,535; and crucially, they can send a much shorter actual payload (say 1 byte), leading to a buffer overread, the contents of which are then returned to them through *public output* on lines 29–32. By default, LeakFuzzer uses a security policy which labels any internal program memory that should not be returned to *public* as *secret*; in this particular case, the contents of the response header, payload from `packet`, and padding bytes should be returned to *public* in the response packet, but anything else should not be.

The contents of the buffer overread will be internal program memory; whatever happens to be allocated in the memory following `packet`. In a freshly brought up system, as the fuzzing harness is, it is likely that this memory has not yet been used and will contain 0s; in a long running server application, this memory could contain secrets such as login request details or other confidential information. Note that OpenSSL is a library for allowing secure communications, and as such is built in to many server-side applications. In the case of a fuzzing harness whereby internal program memory is either zeroed or does not change between executions (due to program state getting reset), sending a malformed packet that triggers the Heartbleed bug many times would not result in any observable difference in output. This is where the LeakFuzzer's aforementioned techniques for handling invalid memory reads come into play.

The `packet` variable is allocated on the heap by `OPENSSL_malloc`, the OpenSSL project's custom wrapper for `malloc`. The `malloc` call within the wrapper is replaced at compilation time by LeakFuzzer, meaning that an extra 8-bytes are allocated and all of the allocated bytes are populated with values generated from the *secret* input that LeakFuzzer generated. Now when a malformed packet triggering Heartbleed is provided as *public* input,


Sent Packet (*public* input)

0	1	2	3	4	5	6
0x01	0x00	0x09	0x74	0x65	0x73	0x74
Packet type: 1	payload_length: 9	't'	'e'	's'	't'

`packet` contents at line 1 without LeakFuzzer approach to memory seeding

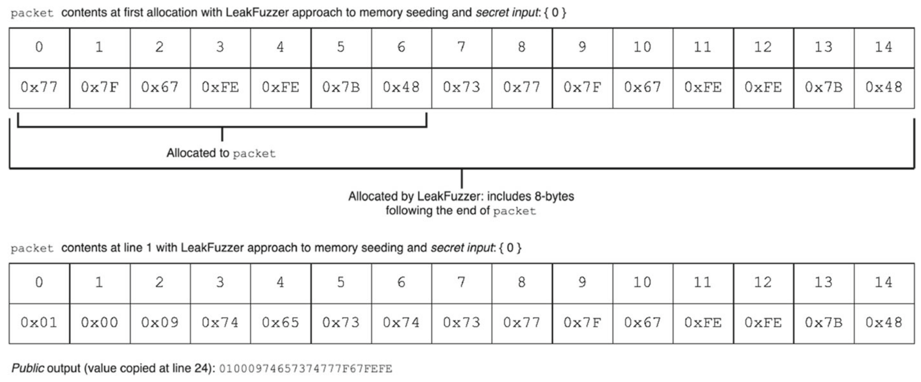
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x01	0x00	0x09	0x74	0x65	0x73	0x74	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Public output (value copied at line 24): `010009746573740000000000`

Fig. 5 An example packet triggering Heartbleed. The top table shows the internal format of the heartbeat request packet as sent by the client. The bottom table shows the memory layout of the allocated `packet` buffer from line 1 of the code listing above; note that the memory after `packet` has not yet been allocated to anything and is thus zeroed. Finally, the bottom line of text shows the output contents sent back to the client (i.e. the output) in hex string form; no matter how many times the fuzzing harness is ran, this output will not differ

the corresponding *public* output is affected by the values of those 8-bytes due to the buffer overread. Producing a hypertest consisting of two inputs with matching *public* parts and differing *secret* parts will demonstrate the insecure flow, as the outputs now differ.

From the above figures, we can see that the pair of tests: `{ public: 01000974657374, secret: 0 }` and `{ public: 01000974657374, secret: 22 }` deterministically produce different outputs `01000974657374777F67FEFE` and `01000974657374917BFFD9F2` respectively. As Fig. 5 shows, without LeakFuzzer's approach to memory handling, we would not be able to observe any difference in output (Figs. 6 and 7).


`packet` contents at first allocation with LeakFuzzer approach to memory seeding and *secret input*: { 0 }

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x77	0x7F	0x67	0xFE	0xFE	0x7B	0x48	0x73	0x77	0x7F	0x67	0xFE	0xFE	0x7B	0x48

Allocated to packet
Allocated by LeakFuzzer: includes 8-bytes following the end of packet

`packet` contents at line 1 with LeakFuzzer approach to memory seeding and *secret input*: { 0 }

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x01	0x00	0x09	0x74	0x65	0x73	0x74	0x73	0x77	0x7F	0x67	0xFE	0xFE	0x7B	0x48

Public output (value copied at line 24): `01000974657374777F67FEFE`

Fig. 6 The top table shows the memory layout of `packet` when allocated by LeakFuzzer, note the extra 8 bytes allocated after the end of packet and the repeating 8-byte pattern that has been used to initialise the memory. The bottom table shows the memory layout after it has been populated with the contents of the sent `packet`; note that bytes 0-6 are unchanged from Fig. 5, but 7-14 are now non-zero. The program output now differs compared to Fig. 5; but does not change if the fuzzing harness is reran with the same *secret* input (0)

`packet` contents at first allocation with LeakFuzzer approach to memory seeding and *secret input*: { 22 }

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x7B	0xFF	0xD9	0xF2	0x1F	0xF3	0x86	0x91	0x7B	0xFF	0xD9	0xF2	0x1F	0xF3	0x86

`packet` contents at line 1 with LeakFuzzer approach to memory seeding and *secret input*: { 22 }

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x01	0x00	0x09	0x74	0x65	0x73	0x74	0x91	0x7B	0xFF	0xD9	0xF2	0x1F	0xF3	0x86

Public output (value copied at line 24): `0100097465737491 7BFFD9F2`

Fig. 7 The top table again shows the memory layout of `packet` when allocated by LeakFuzzer, this time a different *secret* input, 22, was provided and hence a different repeating 8-byte pattern generated. The bottom table shows the memory layout after it has been populated with the contents of the sent `packet`; again bytes 0-6 are unchanged from the previous figure, but 7-14 differ. The program output differs compared to Fig. 6 due to the different *secret* input values (22 and 0)

5 Evaluation and Results

Having described LeakFuzzer and how it works, in this section we evaluate how well it performs against faults that cause insecure flows in the benchmark software suite we have assembled. We assess the usual questions of efficacy and efficiency. We then evaluate how well its performance compares to a representative sample of available tools that can be used to detect insecure flows.

5.1 Research Questions

RQ1: How many known insecure flows in our SUTs are discovered by LeakFuzzer?

We seek to answer efficacy permissively, by the percentage of SUTs in which the known error is found in at least one of the 20 24-hour runs.

RQ2: In what proportion of runs does LeakFuzzer detect insecure flows within a standard 24-hour fuzzing budget?

Here we seek to determine the efficiency of LeakFuzzer by considering the mean time to discovery and the proportion of runs that detect leakage of confidential information within a standard fuzzing campaign budget of 24 hours.

RQ3: How efficacious and efficient is LeakFuzzer in comparison with existing practical approaches that can detect insecure flows?

We compare with three tools: MemorySanitizer, DataFlowSanitizer, and the C Bounded Model Checker (CBMC). We compare the results using the same metrics as used in RQ1 and RQ2 for all three tools. Additionally we look at measurements of memory usage.

First we discuss the creation of the benchmark suite.

5.2 Secure Information Flow Faults (SIFF) Benchmark Suite

No benchmark suite for insecure flows in C and C++ was available to our knowledge. Hence we created SIFF, a secure information flow faults repository that contains programs together with fuzzing harnesses, initial fuzzing seeds and example insecure flow exposing hypertests.

Information flow control issues are difficult to find in known issue databases due to the language used to describe them. Typically they are referred to as *information leaks*, however the word *leak* is overloaded and could refer to memory leaks, resource leaks or confidential information leaks, with the former two being more prevalent in C and C++ programs. In total we have included 12 distinct programs, of which 9 are taken from confirmed CVE reports and others are example programs taken from existing work. Implicit and explicit flows are represented in the program suite as well as memory management and design issues. It also has examples of different sized SUTs and a mixture of kernel and user space programs. In addition, the SUTs have variety in input types including byte strings, human readable formats and SQL queries. The programs taken from CVE reports span errors from 2007 to 2022. We have taken all of the programs containing CVEs from the work using CBMC to detect information leaks (insecure flows)(Heusser and Malacaria 2010). The repository is publicly available for reuse in future research.

A fuzzing harness has been constructed for each individual program. The OpenSSL-1.0.1f fuzzing harness was modified from the version contained in fuzzer-test-suite (Google 2016). Additionally, when testing using AFL++ based systems, the PostgreSQL program uses the AFL++ Grammar Mutator ((h1994st) 2020) plugin to generate SQL statements from a provided grammar. This is because the standard mutation engine struggles to generate valid statements even when provided with many seeds (Table 1).

All programs are written in C with the exceptions of Banking, Password Check and Reviewers; these were originally written in Java and manually converted to C++.

Program Design Errors

Of the 12 programs, six contain insecure flows caused by program design error, where a flow logic error causes the leakage. For these programs, there is explicit *secret* input, *public* input and *public* output; the bug report allows us to construct a security policy for them.

5.2.1 Memory Mismanagement Errors

Four of the programs contain memory related errors and all are detected by LeakFuzzer. For these we use a security policy whereby all user input is labelled *public*, all output is labelled *public*, and any other internal program memory is labelled *secret*. In order to allow us to populate the *secret* internal program memory from input, we use the techniques described in section 4.3. The first of these is appletalk, this is a simplified version of the original bug taken from a previous paper on measuring information leakage (Heusser and Malacaria 2010). This reveals confidential information from stack memory within the Linux kernel, and is caused by failure to initialise all members of a struct before returning the value to userspace. OpenSSL-1.0.1f contains the infamous Heartbleed bug which revealed information by outputting internal program heap memory due to a buffer over-read. RDS (reliable datagram sockets), sigaltstack and tcf_fill_node are also taken from the Linux kernel, with information revealed from stack memory; again caused by failure to initialise all struct members. Finally, sr9700_rx_fixup is taken from a Linux network driver which reveals information from heap memory due to buffer over-read.

5.3 Testing Environment

All tests were run inside Docker containers based on the Ubuntu 20.04 distribution of Linux. Fuzzing experiments were run with 10 fuzzing campaigns in parallel, each bound to a single

Table 1 Program details for members of the Secure Information Flow Faults benchmark suite

Program Name	Flow Type	LoC	CVE Number	Source
appletalk	E + M	110	CVE-2009-3002	Heusser and Malacaria (2010), Klebanov et al. (2013)
Banking	I + PD	150	–	Hamann et al. (2018)
cpuset	E + PD	82	CVE-2007-2875	Heusser and Malacaria (2010)
NetworkManager	E + PD	18,185	CVE-2011-1943	–
OpenSSL-1.0.1f	E + M	279,466	CVE-2014-0160	Google (2016)
Password Check	I + PD	117	–	Hamann et al. (2018)
PostgreSQL	E + PD	905,264	CVE-2021-3393	–
RDS	E + M	94,248	CVE-2019-16714	–
Reviewers	I + PD	145	–	Hamann et al. (2018)
sigaltstack	E + M	141	CVE-2009-2847	Heusser and Malacaria (2010)
sr9700_rx_fixup	E + M	439	CVE-2022-26966	–
tcf_fill_node	E + M	810	CVE-2009-3612	Heusser and Malacaria (2010)

Note that Flow Type is abbreviated using the key: E: explicit flow, I: implicit flow, M: Memory issue, PD: Program design issue

CPU core. Model checking (CBMC) experiments were not run in parallel due to RAM space being a common bottleneck. Tests were run on dedicated servers each equipped with 2 x Intel Xeon E5-2620 v2 processors, making for a total of 12 cores (24 threads) at 2.10GHz, and 128GB RAM. As is common in the evaluation of fuzzers, each benchmark was fuzzed for 24 hours, though this time length does not produce definitive runs on large programs (Klees et al. 2018).

Each experimental setup/run was repeated 20 times. The nature of the mutational engine on the inputs is nondeterministic so different results may occur in different 24 hour runs.

6 Results

6.1 RQ1: How many known insecure flows in the set of benchmarks are discovered by LeakFuzzer?

RQ1: How many known insecure flows in the set of benchmarks are discovered by LeakFuzzer?

For all 12 programs, insecure flows were detected in at least 40% of the runs. This includes all flow types, in particular five of the six program design errors were detected in 95% or more runs. Discussion on the cause of variability follows in RQ2.

LeakFuzzer finds every insecure flow in every program, but not in every run.

6.2 RQ2: In what proportion of runs does LeakFuzzer detect insecure flows within a standard 24-hour fuzzing budget?

Each run used the same set of initial seeds. As can be seen in Table 2, even for the same program, runtime before detecting the first insecure flow varied considerably due to non-deterministic input queue and mutation selection strategies used in AFL++.

Table 2 Results for LeakFuzzer on the SIFF benchmark suite

SUT	% runs flow detected	flow detected time (s)	
		Mean	Std. Dev.
appletalk	100	175.23	250.49
Banking	100	1.48	1.25
cpuset	95	1.16	0.41
NetworkManager	100	228.01	141.43
OpenSSL-1.0.1f	75	40,141.11	21,129.60
Password Check	100	483.65	1,575.00
PostgreSQL	55	32,870.09	19,788.94
RDS	100	3,223.85	8,910.57
Reviewers	100	82.53	87.38
sigaltstack	100	55.42	17.65
sr9700_rx_fixup	40	16,530.35	27,442.28
tcf_fill_node	100	64.86	44.92

The two most complex programs as measured by lines of code—OpenSSL-1.0.1f and PostgreSQL—had the longest discovery times, as would be expected with an exploratory approach such as fuzzing. LeakFuzzer took a surprisingly long time to discover the insecure flow in sr9700_rx_fixup despite containing relatively few lines of code due to the part of the input space that triggers the bug being very small. In this particular case there were 28 edges that could be covered, and in 19 of the 20 runs 17 edges were covered; only 12 were covered in the other run. As might be expected, the insecure flow was not detected in the run that produced less coverage. As the other runs demonstrate, there is not a one-to-one relationship between coverage and the discovery of insecure flows. We see a similar lack of correlation between coverage and insecure flow discovery for both OpenSSL-1.0.1f and PostgreSQL.

It is likely that all fuzzing campaigns would have discovered insecure flows if left to run for long enough; however within the 24-hour time limit imposed here the majority still did.

6.3 RQ3: How does LeakFuzzer compare with existing approaches that could be used to detect insecure flows?

Both MemorySanitizer (MSan) and DataFlowSanitizer (DFSan) were evaluated using a similar fuzzing harness and input seeds. These setups were fuzzed by an unmodified version of AFL++ for 24-hours each to provide a fair comparison against LeakFuzzer. Those programs containing an insecure flow caused by a memory issue were evaluated with MSan, and those containing a program design issue with DFSan (Table 3).

6.3.1 Memory Sanitizer (MSan)

As one of the LLVM sanitizers, MSan can be used in combination with fuzzing to detect uninitialised memory usages. For this reason, it may be able to detect a proportion of the

Table 3 Results for AFL++ in combination with the two sanitizers on the SIFF benchmark suite

SUT	Sanitizer	% runs flow detected	Flow detected time (s)	
			Mean	Std. Dev
appletalk	MSan	0	–	–
banking	DFSan	0	–	–
cpuset	DFSan	100	0.12	0.11
NetworkManager	–	–	–	–
OpenSSL-1.0.1f	MSan	100	2377.47	2732.26
Password Check	DFSan	0	–	–
PostgreSQL	DFSan	0	–	–
RDS	MSan	100	548.71	152.25
Reviewers	DFSan	0	–	–
sigaltstack	MSan	100	330.78	225.23
sr9700_rx_fixup	MSan	0	–	–
tcf_fill_node	MSan	100	101.96	85.25

insecure flows caused by memory mismanagement errors. It is worth noting that the presence of uninitialised memory use does not necessarily imply that the program output will be affected, thus there may not be confidential information revealed by many errors detected by MSan. In order to determine whether a crash detected by MSan 'detected' an insecure flow, the stack trace for each discovered error was manually inspected. If fixing said error would eliminate the flow then it was considered to have been 'detected' in that run. Due to repeats sharing the same executable, each unique crash (i.e. those sharing a stack trace) only needed inspecting once.

The four programs containing insecure flows caused by memory mismanagement – appletalk, OpenSSL-1.0.1f, RDS and sr9700_rx_fixup – were evaluated with MSan, as it can potentially detect these. Of this subset, errors related to the known insecure flow were discovered in OpenSSL-1.0.1f and RDS by MSan. It is likely that the error in OpenSSL-1.0.1f was discovered much more quickly by MSan than LeakFuzzer as many inputs that trigger the buffer over-read do not reach far enough in memory to result in disclosure of confidential information. LeakFuzzer instead requires that there are discernible differences in output so must find those inputs that trigger this behaviour. The error in RDS was also detected more quickly by MSan, though not by such a significant margin.

6.3.2 DataFlowSanitizer (DFSan)

As it provides an implementation of taint analysis, DFSan is more closely related to information flow control than MSan. Where MSan required no modification to the fuzzing harnesses, for DFSan we added labels to the secret inputs and asserted that this label did not propagate to the program output. Any input that resulted in the assertion failing was considered to have discovered the insecure flow.

We had expected that DFSan would be unable to detect the three implicit flows due to the fact that DataFlowSanitizer tracks only *data flow* and not *control flow*; this proved to be the case in all runs. It was slightly less clear whether PostgreSQL would trigger an assertion failure due to the secret values being stored into the database before being fetched in certain queries that expose the insecure flow. As the database might write the values to disk and retrieve from disk, we expect that any taint labels are lost in this process. It is also possible that no query exposing the insecure flow was generated, however it seems unlikely that this would be the case for all 20 runs given that LeakFuzzer succeeded in 11 of its 20 runs. Whatever the cause, the results show that for the setup that we used, DFSan was unable to detect the information flow from secret input to public output. Finally, we observe that the error in cpuset is discovered very quickly by DFSan; this is a simple program, and is explored by the fuzzer very quickly.

The NetworkManager benchmark is a better candidate for detection by DFSan than by MSan, however as both require all dependencies to be compiled with sanitizer flags it was not tested. Attempts were made to resolve all dependencies. However after significant time was spent attempting to do this all the way down the stack too many issues were encountered to consider going any further. Note that the 'banking', 'password check' and 'reviewers' benchmarks required building the C++ standard library from scratch too, though this was managed in reasonable time.

6.3.3 C Bounded Model Checker (CBMC)

Model checking harnesses were created for each of the benchmarks to allow the model checker to verify the presence of an insecure flow. An issue was encountered in building the

Table 4 Results for CBMC on the SIFF benchmark suite

SUT	% runs flow detected	Flow detect time (s)		Exit Cause
		Mean	Std. Dev	
appletalk	100	0.29	0.10	Success
cpuset	100	1.21	0.38	Success
NetworkManager	0	–	–	Segfault
OpenSSL-1.0.1f	0	–	–	Timeout
PostgreSQL	0	–	–	Timeout
RDS	0	–	–	OOM
sigaltstack	100	0.17	0.013	Success
sr9700_rx_fixup	0	–	–	Timeout
tcf_fill_node	0	–	–	Timeout

'banking', 'password check' and 'reviewers' benchmarks, as these are written in C++. This is due to a known issue that CBMC parser "does not handle more modern and template-heavy C++, this means it often runs into problems with parts of the standard library" (tegansb and Schwartz-Narbonne 2020). As a result, the model checker was evaluated on the 7 benchmarks written in C (Table 4).

Three of the smallest benchmarks—appletalk, cpuset and sigaltstack—finished quickly and CBMC managed to find a counterexample falsifying the assertion. In two of these cases, it was able to detect the insecure flow faster than LeakFuzzer on average. LeakFuzzerhad a mean discovery time of 175.23s and 0.17s for appletalk and sigaltstack respectively, compared to 0.29s and 0.17s for CBMC.

CBMC struggled with larger programs as again a 24-hour time limit was imposed, leading to four subjects consistently exiting before the model checker was finished (marked in the table as 'timeout'). Note that tcf_fill_node timed out in our CBMC experiments, whereas it did not in the CBMC approach's original evaluation (Heusser and Malacaria 2010). In the original paper they explain that they use a simplified version of the kernel code but do not

Table 5 Table of results for Memory Usage

SUT	Mean Memory Usage (MiB)		
	LeakFuzzer	AFL++	CBMC
appletalk	2,671.68	28.73	19.13
banking	22,699.78	28.80	–
cpuset	1,926.38	28.74	12.22
NetworkManager	1,622.63	28.27	1,633.14
OpenSSL-1.0.1f	3,196.45	37.75	66,685.88
Password Check	603.52	29.03	–
PostgreSQL	710.54	4.27	7,734.69
RDS	7,433.65	32.30	127,203.04
Reviewers	423.30	36.76	–
sigaltstack	4,927.68	30.50	19.06
sr9700_rx_fixup	5,949.39	205.25	22,477.16
tcf_fill_node	2,928.99	30.50	29,630.58

provide their source code. We instead simply isolate the slice of code without simplifying it, which may explain why our benchmark version does not successfully complete. In the case of RDS, the 128GB of memory available on the evaluation hardware was exhausted causing an out of memory (OOM) early exit. Finally, when attempting the verify NetworkManager, CBMC would exit due to a segmentation fault within the model checker itself.

The formal verification approach was faster in the instances where it managed to resolve however, as has been found in prior works, it fails to scale well enough to work on the larger programs (Table 5).

6.4 Memory Usage Comparison

One of the concerns with storing information about every tested input in LeakFuzzer as described in section 4.1 is the potential memory usage. As can be seen, over the 24 hour runs, memory usage of LeakFuzzer is certainly much higher than AFL++ with the sanitizers, but typically lower than CBMC in the long running test cases. None of the 200 LeakFuzzer runs ran out of memory on the evaluation machine equipped with 128GB RAM.

LeakFuzzer was able to detect the insecure flow in all 12 of the programs in some proportion of the 24-hour runs. The combination of AFL++ with the sanitizers was able to detect insecure flows in five subjects, and CBMC was able to detect flows in just three subjects. Of the 12 programs, insecure flows were detected by all methods in just two: cpuset and sigaltstack. Additionally, LeakFuzzer has been shown to use more memory than AFL++ as expected, however generally less than CBMC, and there were no issues with memory exhaustion in our tests.

7 Conclusions and Future Work

We have presented LeakFuzzer, a fuzzer-based hypertesting approach to detecting leakage of confidential information, and proven its ability to do so on a varied range of benchmarks including seven information leakage related CVEs. It has outperformed a combination of state of the art existing insecure flow detecting techniques, and demonstrated its ability to scale to real-world systems of considerable size. It inherits the advantages of the type fuzzers on which it is based, most notably scalability, coverage and automated input generation. Because it stores more data than a conventional fuzzer, it uses more memory though this was not a problem in our evaluation experiments.

In the future, we hope to extend LeakFuzzer to handle programs written in other programming languages, where sanitizers are not available. Additionally we plan to quantify the flows to allow for application to a broader range of programs, such as password checkers, that need to reveal some amount of confidential information in order to function usefully.

Acknowledgements Daniel Blackwell was supported by UK EPSRC grant no. EP/S022503/1.

Data Availability A replication package including results of the evaluation in this paper is freely available at https://figshare.com/s/fa143b65420ab7ab1e3c.

Declarations

Conflicts of interest The authors declared that they have no conflict of interest.

References

(h1994st) SH (2020) Grammar Mutator - AFL++. https://github.com/AFLplusplus/Grammar-Mutator
4973 C (2014) xxHash. https://github.com/Cyan4973/xxHash
Barthe G, D'argenio PR, Rezk T (2011) Secure information flow by self-composition. Math Struct Comput Sci 21(6):1207–1252
Bell DE, LaPadula LJ (1973) Secure computer systems: mathematical foundations. Technical report, MITRE CORP BEDFORD MA
Bellard F (2005) Qemu, a fast and portable dynamic translator. In: USENIX annual technical conference, FREENIX track, vol 41. Califor-nia, USA, pp 10–5555
Biondi F, Enescu MA, Heuser A, Legay A, Meel KS, Quilbeuf J (2018) Scalable approximation of quantitative information flow in programs. In: International conference on verification, model checking, and abstract interpretation. Springer, pp 71–93
Brennan T, Saha S, Bultan T (2020) Jvm fuzzing for jit-induced side-channel detection. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 1011–1023
Chothia T, Kawamoto Y, Novakovic C (2013) A tool for estimating information leakage. In: Sharygina N, Veith H (eds) Computer aided verification. Springer, Berlin, Heidelberg, pp 690–695
Chothia T, Kawamoto Y, Novakovic C (2014) Leakwatch: estimating information leakage from java programs. In: Kutyłowski M, Vaidya J (eds) Computer Security - ESORICS 2014. Springer, Cham, pp 219–236
Clark D, Hunt S, Malacaria P (2007) A static analysis for quantifying information flow in a simple imperative language. J Comput Secur 15(3):321–371
Clarkson MR, Schneider FB (2008) Hyperproperties. In: 21st IEEE Computer security foundations symposium
Denning DE (1976) A lattice model of secure information flow. Commun ACM 19(5):236–243
Fioraldi A, Maier D, Eißfeldt H, Heuse M (2020) {AFL++}: combining incremental steps of fuzzing research. In: 14th USENIX workshop on offensive technologies (WOOT 20)
Godefroid P, Levin MY, Molnar D (2012) Sage: whitebox fuzzing for security testing. Commun ACM 55(3):40–44
Goguen JA, Meseguer J (1982) Security policies and security models. In: 1982 IEEE symposium on security and privacy, pp 11–11. https://doi.org/10.1109/SP.1982.10014
Google (2011) sanitizers. https://github.com/google/sanitizers
Google (2016) fuzzer-test-suite. https://github.com/google/fuzzer-test-suite
Google (2019) afl-based-fuzzers-overview.md. https://github.com/google/fuzzing/blob/master/docs/afl-based-fuzzers-overview.md
Hamann T, Herda M, Mantel H, Mohr M, Schneider D, Tasch M (2018) A uniform information-flow security benchmark suite for source code and bytecode. In: Nordic conference on secure IT systems. Springer, pp 437–453
He S, Emmi M, Ciocarlie G (2020) ct-fuzz: fuzzing for timing leaks. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, pp 466–471
Heusser J, Malacaria P (2010) Quantifying information leaks in software. In: Proceedings of the 26th annual computer security applications conference, pp 261–269
Hocevar S (2007) zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf
Kinder J (2015) Hypertesting: the case for automated testing of hyperproperties. In: 3rd Workshop on hot issues in security principles and trust (HotSpot)
Klebanov V, Manthey N, Muise C (2013) Sat-based analysis and quantification of information flow in programs. In: Quantitative evaluation of systems: 10th international conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings 10. Springer, pp 177–192
Klees G, Ruef A, Cooper B, Wei S, Hicks M (2018) Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pp 2123–2138

Mesecan I, Blackwell D, Clark D, Cohen MB, Petke J (2021) Hypergi: automated detection and repair of information flow leakage. In: 36th IEEE/ACM International conference on Automated Software Engineering (ASE)

Mesecan I, Blackwell D, Clark D, Cohen MB, Petke J (2022) Keeping secrets: multi-objective genetic improvement for detecting and reducing information leakage. In: 37th IEEE/ACM international conference on automated software engineering, ASE

Metzman J, Szekeres L, Simon L, Sprabery R, Arya A (2021) Fuzzbench: an open fuzzer benchmarking platform and service. In: Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 1393–1403

Miller BP, Fredriksen L, So B (1990) An empirical study of the reliability of unix utilities. Commun ACM 33(12):32–44

Moroz M (2019) google/AFL. https://github.com/google/AFL/blob/master/docs/status_screen.txt

Nilizadeh S, Noller Y, Păsăreanu CS (2019) Diffuzz: differential fuzzing for side-channel analysis. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp 176–187

Noller Y, Tizpaz-Niari S (2021) Qfuzz: quantitative fuzzing for side channels. In: Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis, pp 257–269

Phan Q-S, Malacaria P, Tkachuk O, Păsăreanu CS (2012) Symbolic quantitative information flow. ACM SIGSOFT Softw Eng Notes 37(6):1–5

Sabelfeld A, Myers AC (2003) Language-based information-flow security. Sel Areas Commun 21(1):5–19

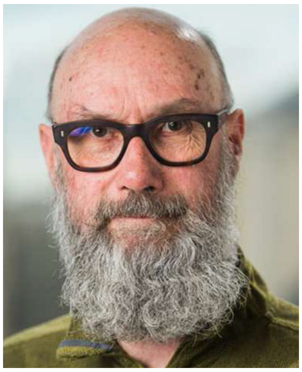
Shen Z, Roongta R, Dolan-Gavitt B (2024) Drifuzz: harvesting bugs in device drivers from golden seeds

tegansb, Schwartz-Narbonne D (2020) github.com - diffblue/cbmc - Parsing Errors when compiling C++ #5489. https://github.com/diffblue/cbmc/issues/5489

Zalewski M (2014) american fuzzy lop (2.52b). http://lcamtuf.coredump.cx/afl/

Daniel Blackwell

Ingolf Becker



David Clark