HyperGI: Automated Detection and Repair of Information Flow Leakage

Ibrahim Mesecan
Iowa State University
Ames, Iowa, USA
imesecan@iastate.edu

Daniel Blackwell
University College London
London, UK
daniel.blackwell.14@ucl.ac.uk

David Clark
University College London
London, UK
david.clark@ucl.ac.uk

Myra B. Cohen
Iowa State University
Ames, Iowa, USA
mcohen@iastate.edu

Justyna Petke
University College London
London, UK
j.petke@ucl.ac.uk

Abstract—Maintaining confidential information control in software is a persistent security problem where failure means secrets can be revealed via program behaviors. Information flow control techniques traditionally have been based on static or symbolic analyses — limited in scalability and specialized to particular languages. When programs do leak secrets there are no approaches to automatically repair them unless the leak causes a functional test to fail. We present our vision for HyperGI, a genetic improvement framework that detects, localizes and repairs information leakage. Key elements of HyperGI include (1) the use of two orthogonal test suites, (2) a dynamic leak detection approach which estimates and localizes potential leaks, and (3) a repair component that produces a candidate patch using genetic improvement. We demonstrate the successful use of HyperGI on several programs which have no failing functional tests. We manually examine the resulting patches and identify trade-offs and future directions for fully realizing our vision.

Index Terms—information flow leakage, genetic improvement

I. INTRODUCTION

The problem of software accidentally leaking confidential information is longstanding [1], much researched [2], and remains an ongoing problem [3]. Its ubiquity and problematic nature has led to high profile security failures such as the famous Heartbleed Bug [4]. The verification research community has extensively studied ensuring Information Flow Control (IFC) as part of the programming process over many decades [5], [6]. IFC is the problem of guaranteeing that a software and a security policy pair satisfy a security property.

As security properties are safety properties most research into IFC has been via verification tools and static or symbolic analyses [7], [8]. While dynamic approaches are not unknown [9], [10] they have been comparatively neglected until recent years. Contemporary software is often large and getting larger [11] and the recent rapid development in the ability of fuzzers to detect security related problems in software is causing a rethink about the value of dynamic approaches and their big advantages in scalability and flexibility [12]. IFC has lacked significant uptake in industry, a significant exception being the SEL4 microkernel [13]. Rather, the emphasis has been on discovering and patching exploitable security vulnerabilities. However, detecting information leaks can not only detect errors in code’s flow logic but also functional errors that lead to leaks, such as memory leaks and buffer overflows [14].

In recent work, Mechtaev et al. demonstrated that they could automatically repair the Heartbleed Bug [15]. However, we caution that this is a special case of IFC, where the program can be made to crash when the safety property is violated. We cannot expect this to hold in general as we demonstrate later.

In this paper we take a fresh look at IFC and ask if we can use a dynamic approach to both detect and repair this important type of security bug. We make two assumptions. First, we cannot assume an IFC will cause the program to fail. Second, we realize that there could be a trade-off between maintaining the original program semantics and removing the information flow leakage. We propose an end-to-end framework called HyperGI. HyperGI, takes a program and a security policy and tests (technically, hypertests) the program for evidence that it leaks and, if it does, estimates the size of the leak. Then HyperGI uses Genetic Improvement [16] to automatically repair the leak while attempting to minimise changes to the program semantics. While the concept of hypertesting programs has been around at least since Kinder’s work [17], it has been little explored in the software engineering community. One recent effort is CT-Fuzz where the hypertest oracle is observing timing and control flow path differences [18]. The strong novelty in our approach is the use of quantified information flow estimates in leak repair, combined with more traditional test cases to ensure functionality invariance.

We have implemented a prototype of HyperGI and apply this to three programs (two of which reported security vulnerabilities used in prior research). We demonstrate that we can reduce the leakage while retaining most of the program functionality. However, we not only identify the need for a multi-objective approach, we note several key aspects of future work needed to fully realize HyperGI, such as the
identification of quality test suites for IFC repair. The contributions of this work are:

- A framework for dynamically detecting, quantifying and repairing information leakage;
- A prototype implementation and first case study to demonstrate its potential.

In the next section we present HyperGI along with background. We then present our study in sections III and IV.

II. HYPERGI

Fig. 1. Overview of HyperGI which consists of three stages

Figure 1 shows a high-level overview of HyperGI. We start with a program (possibly) containing a leak and first generate two types of test sets, Hypertests and Functional tests. We then use a dynamic analysis with just the Hypertest test suite to localize the area of leakage in the program. We then use genetic improvement, using both test suites to iteratively improve the program. We describe each step in more detail next, but first we provide an overview of noninterference and hyperproperties which are fundamental to realizing HyperGI.

A. Preliminaries

A property of program executions can be associated with a partition of the set of all executions into those that have the property and those that do not. Some properties cannot be expressed using an individual execution; rather they require two or more. An example is the noninterference property which states in any software system in which users are divided into groups with different information security access privileges, low security users should not be aware of the actions of high security users [19]. We focus on input/output noninterference for imperative, deterministic programs [2]. There are two groups of users, high security and low security. The states of the program are partitioned between high and low users and low users cannot control inputs to or read from high users’ variables. A program contains the input/output noninterference property iff, for every pair of initial states with the same values of all low variables, but different values for high variables, the post execution states also have the same observable values, i.e., no information about the high variables is leaked. In practice, leaks occur when data from memory is revealed as in a buffer overflow and/or when there is data dependency in control flow.

We have created an exemplar Triangle program to demonstrate. It accepts 3 integers representing the length of each side of a triangle. The first side is a secret value. It returns the type of triangle (isosceles, scalene or equilateral).

TriangleType typeOf(int high, int low1, int low2){
  if (h == low1 && low1 == low2)
    return EQUILATERAL;
  else if (high==low1 || high==low2 || low1==low2)
    return ISOSCELES;
  else
    return SCALENE;
}

Due to the if statements comparing the secret side, any return value potentially reveals some information about the secret. The input \{(high=?, low1=3, low2=4)\} returning SCALENE indicates that the secret value high was neither 3 nor 4, but is only observable if a second test using a different secret returns a different value. If the input \{(high=?, low1=3, low2=4)\} returned ISOSCELES, then the user can deduce that the secret is either 3 or 4. Observing different outputs for the same low inputs but different secret is a violation of the noninterference property. To detect this type we need to use hypertests. Each hypertest is a set of inputs that satisfy the initial states specification of noninterference: the low part of the initial states are the same and the high parts are different. We use the notion of Quantified Information Flow (or QIF) to measure leakage.

B. Stage 1: Test Generation and Leak Detection

In the Triangle program there are two (possibly competing) notions of correctness, functional and noninterference. This program is functionally correct (assuming it accepts only valid triangle inputs); program repair techniques cannot help since there are no failing tests. A set of hypertests may expose the leakage, but fixing the leak can impact functional correctness. A key feature of HyperGI is the use of two independent test suites, one used to test correct program semantics and a hypertest suite used to measure information leakage.

Generating hypertests is challenging since the input space for finding inputs that expose noninterference may be enormous. HyperGI uses a binary search-like algorithm to solve this problem. It simultaneously detects noninterference violations and generates tests to maximize the information leakage. It starts by halving the input space and selects a number (parameter) of low inputs from each half. It then runs the program with a number of executions (another parameter) which alters only the high input(s). Last it checks the resulting output (or data in memory, depending on the security policy) and measures if the same/different values are returned. It builds a priority queue to store hypertests that detect leakage (prioritized by high to low leakage). At each iteration, HyperGI chooses the half of the input space with the largest overall leakage (across all inputs) and repeats on that half. When complete, if it detected a leak, it also has a set of hypertests that reveal it.

1) Quantified Information Flow (QIF): Based on Denning [1], Clark et al. constructed the first program analysis using an information theoretic framework to measure the size of leaks [20]; when a leak is of size zero, noninterference holds. Both
the program language security and machine learning communities have extensively researched estimating information quantities [21], [22]. Bounding QIF can be expressed as a hyperproperty [7], [23] and has been shown to be PSPACE-hard to verify for exact values [24]. HyperGI uses comparative estimations of entropy aiming to reduce the size of the leak to 0. We briefly sketch the mathematical framework.

Given a random variable, the entropy of the random variable is a statistic of its underlying probability distribution that captures the quantity of disorder in the distribution.

**Definition 1:** Let $X$ be a random variable, let $x$ range over the events of $X$, and let $p(x)$ be the probability distribution of $X$. The entropy of $X$, $\mathcal{H}(X)$, is defined as follows:

$$\mathcal{H}(X) = -\sum_{x \in X} p(x) \log p(x)$$

where logs are usually base 2 to retrieve a value in bits.

From Clark et al. we give a definition of the leak size [20].

**Definition 2:** Let $(H, L)$ be the joint random variable in the initial states of a program with a security policy $L \subseteq H$, where $H$ represents high security variable values and $L$ the low security variable values. Similarly, $(H', L')$ represents the joint random variable in the final states. The quantity of leakage from $H$ to $L'$, $\mathcal{L}(L')$, is:

$$\mathcal{L}(L') = \mathcal{H}(L'|L)$$

The intuition here is that the only source of information in the program executions is the input state, represented by the random variable $(H, L)$. The entropy in $L'$ can only exist as a result of the program executions on the initial states, so after factoring out entropy due to $L$, any remaining entropy in $L'$ must be due to $H$. It can be shown that, for non-deterministic programs, this is equivalent to the mutual information between $H$ and $L'$ given knowledge of $L$, $\mathcal{I}(H; L'|L)$ [20].

Conditional entropy calculation is cumbersome, the following chain rule can streamline it for joint random variables [25].

**Proposition 1:** Chain Rule for Entropy

$$\mathcal{H}(A, B) = \mathcal{H}(A|B) + \mathcal{H}(B)$$

We then have QIF: $\mathcal{H}(L'|L) = \mathcal{H}(L', L) - \mathcal{H}(L)$.

**C. Stage 2: Leak Localization**

HyperGI uses a dynamic algorithm that iteratively removes each line of the program and calculates the change in QIF of the program with that line removed. Non-compilable programs have a change of zero. It normalizes all of the QIFs (dividing all by the maximum change) and partitions the resulting statements into equivalence classes. Probabilities are assigned to each class which guides the repair towards those statements which are likely to reduce information flow the most.

**D. Stage 3: Repair**

We implemented genetic programming (GP) [26] search on top of an existing genetic improvement framework, PyGGI [27]. Chromosomes are patches to the AST. We use the standard, delete, replace, insert operators, as well as two new operators. Since information flow leakage is highly control flow dependent we added operators to insert new control flow. One creates new |if| statements (using variables from the program) and the other creates a new |for| loop. Statements within the if/for blocks are created by copying existing statements from the target program, or by creating simple assignments between existing program variables. Fitness is one of the essential parts of GI and it guides the search process by measuring how fit the patches are. The HyperGI fitness function combines both the QIF and functional correctness. The fail rate of mutant $k$ is $f_k = (\#failing\ functional\ tests)/(\#functional\ tests)$. And, $l_k$ is the initial program leakage and $l_k/l_0$ is the normalized leakage of mutant $k$ (defined if mutant $k$ compiles and runs). Then, the fitness of mutant $k$ can be defined as:

$$f_k = 0.5 \times l_k/l_0 + 0.5 \times f_r_k.$$  

**III. EVALUATION**

We conducted a feasibility study to understand the potential for HyperGI. We answer the following research questions:

**RQ1** How does HyperGI compare with fuzzing in terms of leak detection?

**RQ2** How well does HyperGI remove information flow leakage while maintaining software functionality?

To answer these questions we conducted a pilot study: we first run fuzzers to gather functional tests; next, we run our binary search to generate hypertests; then, we run GP-based repair to try to decrease leakage; and, finally, we manually analyse generated patches.

We use three C subjects, two of which were used in prior work on statically finding information flow leakage [7] and which are simplified versions of the original programs from the CVE vulnerability database [28], [29]. The third program is one that we wrote to demonstrate the second type of leakage described in this paper, a control-flow based privacy leak. We generate functional and leakage test suites, as described below. We show details of the subjects in Table I, and present security policies for the two new subjects:

<table>
<thead>
<tr>
<th>Subject</th>
<th>Ref</th>
<th>CVE-#</th>
<th># Funct Tests</th>
<th># Hyper Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle</td>
<td></td>
<td>CVE-2009-3002</td>
<td>254</td>
<td>194</td>
</tr>
<tr>
<td>Apple Talk</td>
<td>7</td>
<td>CVE-2007-2875</td>
<td>186</td>
<td>100</td>
</tr>
<tr>
<td>Underflow</td>
<td>7</td>
<td>CVE-2007-2875</td>
<td>186</td>
<td>100</td>
</tr>
</tbody>
</table>

For comparison with existing dynamic approaches, we ran two state-of-the-art fuzzers AFL [12] and LibFuzzer [30] on all subjects to see if we could detect the leakage. Information
leaks due to buffer overflows can often be found with fuzzers as memory leaks or buffer overflows (such as Heartbleed) can be interpreted as crashes via tools such as AddressSanitizer [31]. We ran 5 runs of each (with different random starting seeds) for 24 hours for each program. To increase test input diversity, we also ran each fuzzer 20 times for 2 hours with randomly generated input seeds. For functional testing we used all tests from all 50 runs (= 25 x 2 fuzzers) for each program with duplicates removed (see Table I for final counts). To generate hypertests, we ran our binary search on each subject, as described in Section II-B.

To fix detected leaks, we ran GP for 25 epochs, each with 50 generations and a population of 32. The target fitness was 0.0 (the program ends if it reaches this fitness) and we examined the best solution (or the stopping solution in case the program ended before 50 generations). We examined all 75 patches (= 25 x 3 programs) for their quality and categorized them based on how well they fix the leak and/or retain the functional correctness of the program. For atalk and underflow we have developer patches from github for reference.

IV. RESULTS

Both fuzzers ran to time limits without finding any crashes or leakage. In contrast, our binary search approach generated hypertests which were able to detect information flow leakage in all three programs. This indicates that these types of leakage cannot always be found by conventional fuzzing. After 320 hours they were unsuccessful while the binary search detected each leak in less than two hours. In answer to RQ1: Our proposed binary search strategy was able to detect leaks for all three programs, in contrast to traditional fuzzing.

Table II shows the results of applying HyperGI to reduce information flow leakage. The first column shows the initial detected information flow leakage (i.e., $l_k$). As we can see, all three programs have a leak with QIF ranging from 13.00 in atalk to 0.83 bits in the triangle program. For each subject we show the median, average and standard deviation of the raw (not normalized) QIF (i.e., $l_k$), the test failure ratio, (i.e., $f_{\text{TR}}$), and the overall program fitness (where 0 is the optimal fitness). For atalk we were able to reduce the QIF to zero while maintaining program functionality (0 failed tests). For the other two subjects we see a trade-off. In fact, in the triangle case we can never create a semantically equivalent and non-leaking program.

<table>
<thead>
<tr>
<th>Patch Quality</th>
<th>triangle</th>
<th>atalk</th>
<th>underflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantically-equivalent fix</td>
<td>– 76%</td>
<td>28%</td>
<td></td>
</tr>
<tr>
<td>Leakage reduction, no functionality loss</td>
<td>0%</td>
<td>24%</td>
<td>0%</td>
</tr>
<tr>
<td>No leakage, but functionality loss</td>
<td>64%</td>
<td>0%</td>
<td>28%</td>
</tr>
<tr>
<td>Leakage reduction, loss of functionality</td>
<td>24%</td>
<td>0%</td>
<td>4%</td>
</tr>
<tr>
<td>No improvement over the original program</td>
<td>8%</td>
<td>0%</td>
<td>23%</td>
</tr>
<tr>
<td>Introduced indeterministic behavior</td>
<td>4%</td>
<td>0%</td>
<td>16%</td>
</tr>
</tbody>
</table>

We now turn to Figure 2. This shows the normalized QIF (over the pre-patch QIF) versus the functional test failure ratio. We plot all 25 epochs for each program. For atalk (black), all 25 data have the same value thus showing a single point (all tests pass and QIF=0). For underflow (blue) we have two general patterns: either the leakage is 0 and 97% of functional tests are failing; or the leakage is 1 (no improvement), but the functionality is retained. This suggests the need for multi-objective optimization. For the triangle program (red dots) we see a wider range of points. Noticeably, there are no points which have retained all of the program functionality.

We manually verified the quality of generated patches. For triangle we don’t have a developer patch, but for the other two we used that as a baseline. Table III shows this data. For two programs we find patches that are semantically-equivalent to the developer one (for 76% of epochs for atalk and 28% of epochs for underflow). We also reduce leaks in the remaining 24% of atalk runs, without loss of functionality. For 84% of cases for triangle and 32% of cases for underflow we improve leakage at the cost of functionality.

A patch that reduces leakage, but breaks some program functionality can still be acceptable. The semantically-equivalent patches for underflow indeed fail our functional tests — that is because to remove leakage the developers had to amend functionality of the program. This trade-off was not necessary for atalk.

We found that some patches introduced nondeterministic behavior. In those cases our tests became flaky, and the QIF
could potentially increase, as in the case of triangle in Figure 2. Another example, from the underflow experiment, is a patch that removes a return statement, hence the program’s output became undefined and thus returned different results each time it was run. In answer to RQ2, HyperGI was able to find patches semantically-equivalent to developer fixes. It found patches reducing leakage in all three programs.

V. CONCLUSIONS AND FUTURE WORK

We propose HyperGI, a framework for dynamically detecting, quantifying and fixing information flow leaks using lightweight dynamic analysis, hypertesting, and genetic improvement. HyperGI was able to reduce information leakage in three programs, producing fixes semantically-equivalent to developer patches. We see a trade-off between quantified information flow and program functionality. As future work we will build HyperGI as a multi-objective framework. We found in some cases, e.g. atalk, despite obtaining a zero (or optimal) fitness, some patches were overfitted. Finding a good set of test cases, hypertests in particular, is important future work. Finally, we will run larger scale experiments on more subjects.

REFERENCES