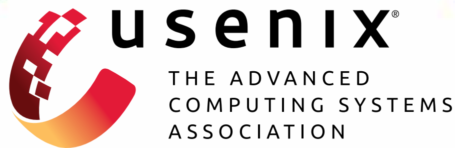


Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code

Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu, *Northeastern University*

https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code

Mansour Ahmadi
Northeastern University
Mansosec@gmail.com

Reza Mirzazade Farkhani
Northeastern University
mirzazadefarkhani.r@northeastern.edu

Ryan Williams
Northeastern University
williams.ry@husky.neu.edu

Long Lu
Northeastern University
l.lu@northeastern.edu

Abstract

Probabilistic classification has shown success in detecting known types of software bugs. However, the works following this approach tend to require a large amount of specimens to train their models. We present a new machine learning-based bug detection technique that does not require any external code or samples for training. Instead, our technique learns from the very codebase on which the bug detection is performed, and therefore, obviates the need for the cumbersome task of gathering and cleansing training samples (e.g., buggy code of certain kinds).

The key idea behind our technique is a novel two-step clustering process applied on a given codebase. This clustering process identifies code snippets in a project that are functionally-similar yet appear in inconsistent forms. Such inconsistencies are found to cause a wide range of bugs, anything from missing checks to unsafe type conversions. Unlike previous works, our technique is generic and not specific to one type of inconsistency or bug. We prototyped our technique and evaluated it using 5 popular open source software, including QEMU and OpenSSL. With a minimal amount of manual analysis on the inconsistencies detected by our tool, we discovered 22 new unique bugs, despite the fact that many of these programs are constantly undergoing bug scans and new bugs in them are believed to be rare.

1 Introduction

Using machine learning techniques to detect software bugs has been studied extensively. Existing works generally follow the same high-level idea: training models on a large set of known bugs and then using the trained models for detecting similar bugs in the wild. This line of work, including [13, 20, 27, 28], has been shown to be largely effective at catching known bugs. However, these "learn-from-bugs" type of detection techniques face two limitations when used in practice. First, they generally require large datasets of known bugs for training, which can be difficult or impractical to collect and cleanse. Second, the models usually have to be trained on specific types of bugs to achieve good results. Therefore, the training and detection are usually limited to a single bug type (i.e., bug-specific). Moreover, the detection accuracy tends to vary a lot across different bug types.

In this paper, we present a new machine learning-based approach to software bug detection, which does not require external datasets or code samples for training (e.g., code containing known bugs). Instead, it learns from the to-be-checked codebase itself (hence the paper title). It is not limited to any bug types, and it can even detect unknown types of bugs. Our approach is inspired by the observation that many bugs in software manifest as inconsistencies deviating from their non-buggy counterparts, namely the code snippets that implement the similar logic in the same codebase. Such bugs, regardless of their types, can be detected by identifying functionally-similar yet inconsistent code snippets in the same codebase. For instance, from basic bugs such as absent bounds checking to complex bugs such as use-after-free, as long as the codebase contains non-buggy code snippets that are functionally similar to a buggy code snippet, the buggy one can be detected as an inconsistent implementation of the functionality or logic. This observation is more obvious in software projects of reasonable sizes, which usually contain many clusters of functionally-similar code snippets, often contributed by different developers. It is very uncommon for all such snippets to have the same bug (or their developers to make the same mistake).

Our work, named `FICS`, uses a machine learning-based method to detect functionally-similar yet inconsistent code snippets in a given codebase, facilitating the detection of inconsistency-related bugs. We note that

the high-level idea of detecting bugs as deviations from normal code is not new. Previous works adopted this idea for detecting system errors [7], incorrect API usages [14, 32], and other specific types of bugs [15, 24, 30]. However, FICS is significantly different from these works in that: (1) it is not specific to one or a few types of bugs; (2) it does not require any domain expertise about bugs or manually-defined detection heuristics.

Figure 1 shows the high-level workflow of FICS. It starts by extracting code snippets (or Construct, explained shortly) from a given codebase (❶). It then performs a two-step clustering method, which first groups functionally-similar parts of the code (❷) and then detects deviations or inconsistencies among them (❸). Finally, the detected inconsistencies are presented to a human analyst for bug triage (❹).

There are two principal challenges solved by our design of FICS: (1) finding a proper code granularity to effectively capture functionalities and inconsistencies, and (2) making the approach scalable to handle large codebases. Given that security-related bugs and patches are often regional or contained in a sub-function scope [12], we propose an intra-procedural granularity, named Construct, which is defined as a size-configurable sub-graph of an intra-procedural data dependence graph. We show that this granularity is sufficient for capturing code similarities and inconsistencies yet small enough to allow the clustering algorithms to scale to large codebases. Moreover, we employ two graph embedding techniques: (1) *bag-of-nodes*, a coarse-grained graph embedding for the first-step clustering; (2) *graph2vec*, a fine-grained graph embedding for the second-step clustering. They enable a sufficiently accurate comparison of Construct similarity and inconsistency at scale.

Our work makes the following contributions:

- We present FICS, the first bug-generic inconsistency-based bug detection method. It uses a two-step clustering method to detect functionally-similar yet inconsistent code snippets. The detection operates on Constructs, a size-configurable graph representation of sub-function code, specially-defined to facilitate code similarity comparison. FICS also uses two new graph embedding techniques, one for each clustering step, which together make the tool sufficiently accurate and scalable to large codebases;
- We used FICS to scan five popular open source projects, including QEMU and OpenSSL. Despite that some of these projects are considered well-tested, we discovered 22 new unique bugs with minimal manual effort. All of the bugs have been confirmed by their developers and later fixed by either our pull requests or the developers. In addition, our approach also found 95 code smells like redundant checks;
- Due to the lack of a standard benchmark that would allow systematic evaluation of inconsistency detection tools, we propose a novel open-source benchmark, named iBench, as another contribution of this work. iBench contains 22 known bugs in real software. We further evaluate FICS on iBench and show that FICS can outperform current inconsistency detection approaches.

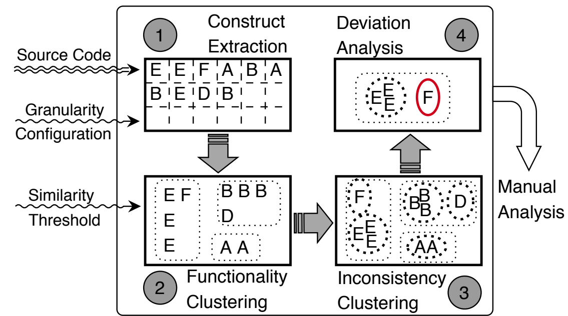


Figure 1: High-level workflow of FICS. After a codebase is divided into smaller pieces, they need to be grouped based on easy-to-learn characteristics efficiently, and then the most discriminated (inconsistent) item can be distinguished by learning from more detailed features.

2 Background

Inconsistency in code has a broad meaning and may refer to inconsistent use of APIs, typecasting, checks, etc. As we aim for a generic code inconsistency detection, we adopt a general definition for inconsistency. We call a set of code snippets inconsistent if their semantics or logic are synonymous, but some parts of their implementation differ in significant ways.

Although not all occurrences of inconsistent code indicate bugs, inconsistent implementations of the same functionality often suggest programmer confusion or mistakes, which in practice often leads to bugs. Previous research [9] has shown success in finding thousands of bugs and not-buggy implementation issues by only tracking down inconsistencies in code clones. Inconsistencies sometimes may indicate critical security bugs even in well-tested codebases written by well-known companies.

Various factors can introduce inconsistencies into a codebase. For example, large software usually has a commensurate number of developers, which sometimes leads to inconsistent implementations of the same functionality during development. Another common reason is when a bug fix is applied only to where the bug was originally discovered, but not to other parts of the code with the same bug. It is also plausible that the same bug will appear again in the future. For example, a similar bug to three missing check bugs–which were found by our system–in LibTIFF

```
787: size_t dukmlen = 0;
 [...]
873: dukmlen = ASN1_STRING_length(ukm);
874: dukm = OPENSSL_memdup(ASN1_STRING_get0_data(ukm), dukmlen);
875: if (!dukm)
 [...]
879: if (EVP_PKEY_CTX_set0_dh_kdf_ukm(pctx, dukm, dukmlen) <= 0)
 [...]
 Missing 'OPENSSL_free(dukm)'
```

(a) crypto/dh/dh_ameth.c, `dh_cms_encrypt` function

```
677: size_t dukmlen = 0;
 [...]
729: dukmlen = ASN1_STRING_length(ukm);
730: dukm = OPENSSL_memdup(ASN1_STRING_get0_data(ukm), dukmlen);
731: if (!dukm)
 [...]
735: if (EVP_PKEY_CTX_set0_dh_kdf_ukm(pctx, dukm, dukmlen) <= 0)
 [...]
742: OPENSSL_free(dukm);
```

(b) crypto/dh/dh_ameth.c, `dh_cms_set_shared_info` function

Figure 2: A memory leak in OpenSSL found by `FICS`. The figures show two similar `Constructs` on `dukmlen` variable in two different functions. This could be seen as an inconsistency in the code as there is one buggy and one correct implementations. The red node is missed in `dh_cms_encrypt` function.

had been patched 4 years ago [1] by its developers because they received a crash report from a fuzzer at that time.

An intuitive idea to detect code inconsistencies is based on majority-voting: if there are multiple pieces of code implementing the same functionality, we can generally assume the majority is correct, and any deviation from the majority could be signs of buggy or low-quality code. For instance, we found a bug in OpenSSH by detecting the inconsistencies among the code snippets operating on a `hash` variable , where the key is cleared from memory in three implementations while it is missed in the fourth one.[1]

Although majority-voting assists analysts in identifying buggy code more reliably, it is still possible that a piece of buggy code is similar to only one functionally-similar code snippet in the codebase. We call such cases *one-to-one inconsistencies*. Unlike the majority-voting-based approaches, `FICS` can detect one-to-one inconsistencies. `FICS` takes into consideration the size of the code snippets. If two code snippets (i.e., one buggy and one non-buggy) have nontrivial sizes, multiple operations are performed on the target variable and a small difference/inconsistency between the two snippets can indicate bugs. For example, Figure 2 shows an example of a memory leak `FICS` found in OpenSSL. Although there are only two similar code snippets (e.g., a one-to-one inconsistency), many operations on the `dukmlen` variable in both snippets are the same, and a missing free in one of them makes this one-to-one inconsistency a true bug. Majority-voting-based approaches cannot detect such bugs.

`FICS` is designed to detect code inconsistencies indicative of bugs without being limited to one or a few specific types of bugs. The advantage of our inconsistency-based bug detection is three-fold. First, our detection is bug-generic. It complements existing bug detectors, most of which tend to be specific to certain types of bugs due to the limitation of their heuristics or trained models. Second, our detection does not require any external data or code samples for training, or any domain expertise about bugs and their manifestations. Lastly, our system can detect an inconsistency with or without majority-voting.

[1] https://github.com/openssh/openssh-portable/commit/2d1428b

3 Related Work

ML for Bug Discovery: Machine learning (ML) techniques have been used successfully to model and detect buggy code patterns in different programming languages. Usually, such approaches are divided into two groups, namely supervised and unsupervised learning.

Supervised techniques have been used to model both buggy and non-buggy code patterns. Motivated by its huge success in other domains, deep learning techniques were recently used by researchers for detecting bugs. VulDeePecker [13] applies deep learning techniques, specifically a bidirectional LSTM model, to automatically learn patterns from vulnerable code gadgets. The code gadgets are small parts of the code that are extracted based on program dependency. The main drawback of these approaches is the heavy efforts required for gathering, cleansing, and labeling a large number of training samples.

Without requiring data labeling, unsupervised learning approaches cluster code snippets that are similar to known vulnerabilities and then search for potential variants of the vulnerabilities in each cluster. Yamaguchi et al. [29] proposed a method for assisting security analysts with source code auditing. It can identify vulnerabilities by inspecting only a small fraction of the codebase. A follow-up work [28] introduced a novel representation of source code, using a joint data structure, called a code property graph. This representation draws from ASTs, control flow graphs, and program dependence graphs.

All of the aforementioned works use ML techniques to learn from the known bugs and use the learned models to discover occurrences of modeled bugs. In comparison, our approach does not require training on labeled datasets or bug-oriented clustering. Instead, we apply ML techniques to find functionally-similar yet inconsistent code snippets, which often contain bugs and can be easily verified by developers or testers when presented with both the consistent and inconsistent implementations of the same functionality or logic.

Genius [8] addresses the scalability issue in the existing ML-based bug-finding techniques and further improves search accuracy. It embeds control-flow graphs (CFGs) into high-level numeric feature vectors. Xu et al. [27]

uses the structure2vec technique to extract more accurate embedding from different code snippets and then trained a Siamese network to detect similar bugs. While such approaches used graph embedding to find variants of known bugs in different program versions, we drew inspiration from these works for inconsistency detection and proposed: (1) a new code representation of proper granularity for functional similarity comparison; and (2) adopting two popular graph embedding techniques, namely bag-of-nodes and graph2vec, to model inconsistencies in an accurate and efficient way, which make our approach scalable to large codebases in the real world.

Inconsistency Detection: Engler et al. proposed to analyze bugs as deviant behavior [7], which is a seminal work related to code inconsistency detection. Their approach is to infer developer *beliefs* and then cross-check them with the implementation for contradictions. Bixie [16, 21] is a tool that detects a form of code inconsistency, defined as code fragments outside of any normally terminating execution (e.g., dead code or code making conflicting assumptions). This line of work is by nature rule-based and focused on a few specific types of programming errors, such as assertion violations. In comparison, our ML-based approach is not limited to detecting particular types of bugs.

DejaVu [9] and Jiang et al. [11] proposed generic techniques to detect syntactic inconsistencies in code. These works rely on abstract syntax tree, which is not semantic-aware, to find inconsistent code. This approach and ours share the same high-level idea of utilizing code inconsistency for detecting errors in programs. However, our work is semantic-aware and detects inconsistencies at a deeper level using a novel code construct representation that captures not only abstract syntax but also data and control flows. As a result, these previous work is unable to detect the majority of the bugs that `FICS` can.

Some bug detectors, though not designed to capture general inconsistencies in code, can be viewed as identifying specific types of coding inconsistencies. APIsan [32] infers correct API usages in source code through symbolic execution and semantic cross-checking. Similar to APIsan, AntMiner [5, 14] and NAR-Miner [6] detect API usage inconsistencies. However, instead of using symbolic execution, they mined programming rules (e.g., frequent patterns) from the program dependency graph to detect violations. Chucky [30] detects a specific type of code inconsistency, namely missing checks, which are often indicative of security-related bugs. Chucky uses a rule-based detection method, which requires a list of pre-defined APIs as sinks for its taint analysis. Crix [15] is the most recent inconsistency detection technique on missing checks. It identifies critical variables based on the concept of security checks [24], and then cross-checks the modeled constraints of the peer slices of a critical variable. The notion of a slice (`Construct`) by Crix is a data flow path–not a data flow graph.

The above line of work is semantic-aware and closely related to `FICS`. However, our work overcomes two major limitations of the prior work. First, these works were designed to detect *only* specific types of inconsistencies, such as those related to API usage [5, 14, 32] or sanity checks [15, 30]. They cannot detect other classes of inconsistencies that their design was not modeled after. Moreover, their approaches cannot be easily extended to detect inconsistencies or bugs in a type-agnostic fashion. In contrast, `FICS` provides a generic approach to detecting inconsistency-induced bugs regardless of inconsistency types, thanks to the novel two-step clustering design. Second, the prior approaches need to rely on majority voting to determine inconsistencies because of the limitation in their code construct definitions. The majority voting-based approach cannot detect one-to-one inconsistencies (explained in §2) as `FICS` does. Our graph-based definition of code constructs carries additional context information, which captures one-to-one inconsistency. Besides, our inconsistency detection does not involve opportunistic majority voting. Nonetheless, if a bug is a one-liner or its inconsistency is visible only in a very small code construct, Crix [15] and APISan [32] are better suited to catch it when the bug type matches their detection targets (e.g., a missing check or an API misuse).

4 Design

4.1 System Overview

`FICS` is the first machine learning-based bug detector that learns and identifies code inconsistencies as indicators of bugs. It is agnostic to bug types and more generic than previous inconsistency detectors, which tend to be limited to certain types of bugs.

Figure 3 shows the workflow of `FICS`. `FICS` first compiles a given codebase in C into LLVM bitcode (❶), on which the subsequent analysis and learning steps are performed. It then employs an intra-procedural data-flow analysis (§4.2.1) to extract from each function small code pieces, referred to as `Constructs` (§4.2.2), that represent basic operations or computations within a function. We use such intra-procedural `Constructs` as the inconsistency detection granularity for two reasons. First, most security bugs and fixes tend to be limited within a single function [12]. The difference between a piece of buggy code and its non-buggy counterpart (or its patched version) usually does not extend beyond a function. Second, having `FICS` focused on intra-procedural inconsistencies

makes the analysis scalable to large codebases.

After extracting the `Constructs` (❷–❸), `FICS` abstracts the `Constructs` (❹) to a generic form amenable to the two-step clustering (❺–❽). The first-step clustering (§4.3.1) groups functionally-similar `Constructs` whereas the second-step clustering, zooming in on each group, finds `Constructs` that are inconsistent from the rest in the same group. The two-step clustering is designed to accurately capture functionally-similar yet inconsistent `Constructs` while remaining scalable to large codebases.

In the final step (❾), `FICS` performs a deviation analysis (§4.4), which identifies inconsistencies that are indicative of bugs. The result helps human analysts focus on potential bugs and facilitate bug triage.

4.2 Code Representation and Granularity

Finding a suitable *code representation* and determining a proper *code granularity* are the first two challenges that we solved in order to employ machine learning techniques to effectively identify similar yet inconsistent code. We discuss our choice of the code representation in §4.2.1 and our definition of the code granularity in §4.2.2.

4.2.1 Simplified Program Dependence Graph

Among the existing code representations [4], Program Dependence Graph (PDG), Control Flow Graph (CFG), and Abstract Syntax Tree (AST) are the best-known and widely used for bug discovery [27, 28, 29]. ASTs capture syntactic information of programs. CFGs record possible code paths as well as path conditions. PDGs illustrate data and control dependencies among program statements.

We select PDG as the base to develop a program representation for `FICS`. We choose PDG because it is the most semantically comprehensive among the common program representations, which suits our need for discovering and clustering functional similarities. Moreover, PDGs were originally proposed for the purpose of program slicing [25]. Their sub-graphs naturally capture regional control and/or data dependencies, which serve as ideal primitives for defining `Constructs` or the granularity of inconsistent code (discussed in §4.2.2).

In PDGs, a data dependency edge appears from a program statement to another when an output of the egress statement is an input to the ingress statement. A control dependency edge appears when the evaluation of the egress statement determines whether the ingress statement is reachable during program execution. We derive a code representation for `FICS` by omitting control dependency edges while keeping data dependency edges in PDGs. Furthermore, our representation is intra-procedural and context-insensitive. We call this form of code representation *Data Dependency Graph*, or DDG for short, which is similar to the one used in thin slicing [23].

Our design of DDG is based on the following considerations and trade-offs. First, data dependencies alone are enough to capture the root cause of a wide range of bugs. For instance, missing checks, misuses of APIs, bad castings, and many other types of bugs manifest clearly in a DDG when compared with non-buggy or patched counterparts. By omitting the control dependence edges, DDG allows for much more simplified representations of code, and thus scalable and efficient analysis, without losing important semantics for detecting common bugs. Second, bugs and their patches are often contained in a single function [12], which means the difference between buggy and non-buggy code snippets is observable on intra-procedural DDGs. By limiting the scope of analysis within individual functions, DDG further improves the scalability of `FICS` and allows for analysis of large codebases. Third, being a graph-based representation, DDG lends itself nicely to mature machine learning techniques (e.g., embedding and clustering) for detecting similarities and inconsistencies.

4.2.2 `Construct` Definition

`FICS`'s effectiveness and efficiency also heavily depends on the code granularity at which the similarity and inconsistency analysis is performed. We introduce the concept of `Construct` and show that it is a proper code granularity for our purpose. Informally, a `Construct` is a subgraph of a DDG, which represents a somewhat self-contained subroutine that is part of a function (e.g., processing a parameter to an API call). `Constructs` are extracted in a way conceptually similar to program slicing [25].

Given a DDG, the extraction starts from a specified node (i.e., the root) and traverses the DDG until all subsequent nodes are covered or the `Construct` max-depth is reached. All traversed nodes and edges then form a `Construct`. A *root variable* and the *max-depth* uniquely define a `Construct`. Any variable V used in a function F can be selected as the root variable for extracting a `Construct` C. In this case, the root of C is the node in F's DDG where V is defined or used for the first time (i.e., when V enters F). In other words, C contains all code statements inside F that compute on, or propagate, V.

The max-depth restricts the number of basic blocks that the longest forward path in a `Construct` may contain. It is configurable and can be tuned by `FICS` users to limit the depth of `Constructs`, and therefore, control the max size of code on which similarity and inconsistency are determined. By default, the max-depth is set to infinite (i.e., no limit on the depth of `Constructs`). As a result, a `Construct` contains all nodes and edges in a DDG reachable from the root. We call such a depth-unlimited `Construct` a `full-Con`. When the max-depth is set to

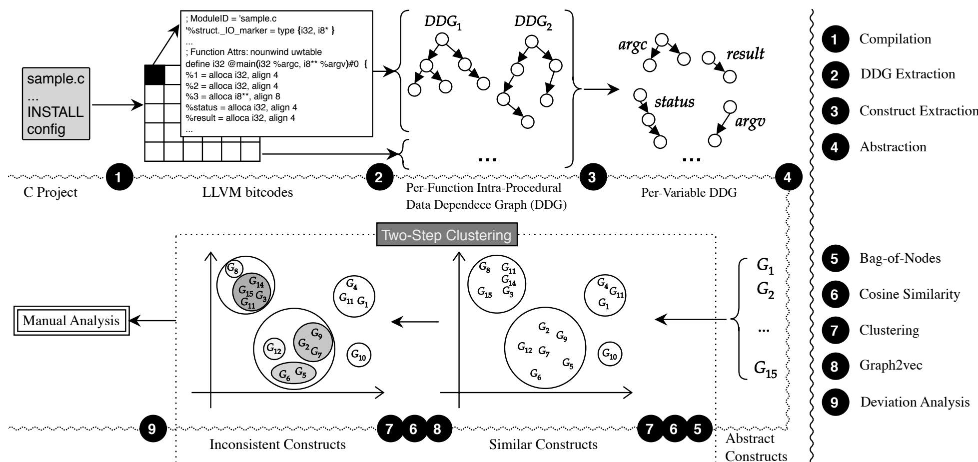


Figure 3: `FICS`'s workflow contains 9 steps. It extracts `Constructs` from intra-procedural data dependence graphs. Via a two-step clustering process, `FICS` groups similar `Constructs` and then finds inconsistencies within such groups. Eventually, `FICS` outputs those functionally-similar-yet-inconsistent `Constructs` that are likely to be bugs, which human analysts can easily triage.

a finite number n, a `Construct` cannot have more than n basic blocks on any forward path. Such `Constructs` are referred to as `n-Con`. For example, when $n = 1$, a `Construct` (i.e., `1-Con`) contains only 1 basic block, where the root is; when $n = 2$, a `Construct` (i.e., `2-Con`) can at most contains 2 basic blocks along a forward path. An example of a `full-Con` is shown in Figure 4b, which is extracted from Figure 4a. An example of the extracted `1-Con` is shown in Figure 4c.

By defining `Constructs` this way, we generalize the problem of detecting code inconsistency into the problem of finding similar and inconsistent operations/computations on individual data variables. This generalization allows inconsistencies to be detected in a generic fashion (i.e., not tied to a specific type of code inconsistency) and at a relatively small code granularity.

For each function in a program, `FICS` extracts the `Construct` for every parameter and local variable of the function and every global variable reachable to the function. In addition, `FICS` performs `Construct` abstraction (❹ in Figure 3), which serves two goals: (1) removing certain syntax information that is useless for `FICS` and can negatively affect the similarity clustering; and (2) further minimizing `Constructs` for more efficient clustering. To abstract `Constructs`, we preserve only the variable types for each program statement and remove all variable names and versions[2]. We also eliminate all constants and literals from program statements. One could argue that integer literals in conditionals (e.g., `icmp`) might be useful for detecting certain bugs (e.g., a comparison is done with 0 instead of EOF or -1). However, based on our observation, considering integer literals in `icmp` and similar instructions leads to significantly more reported inconsistencies that are not true bugs. Finally, abstract `Constructs` are then used as the input to the two-step clustering.

To understand the potential negative impact of forgoing literals in our abstraction, we investigated a class of bugs, namely *off-by-one* errors, that one may expect our inconsistency detection to miss. In fact, as shown in §5.2, `FICS` can detect 7 out of 11 such bugs in the tested codebases. This is because, for the majority of off-by-one errors, the difference/inconsistency between the buggy and non-buggy/patched code lies in the comparison operators, rather than the literals.

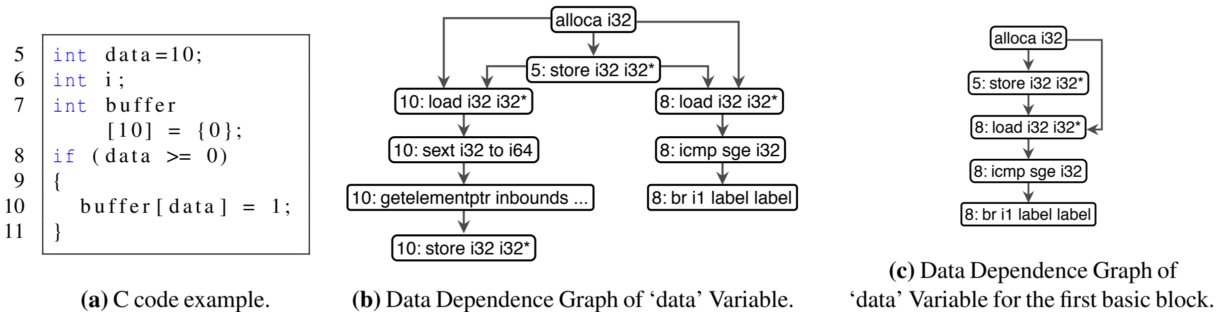
4.3 Two-step Clustering

We design a two-step clustering procedure to first group similar `Constructs` and then identify inconsistent `Constructs` or outliers in each group. The high-level idea can be easily explained using a "two-lense" analogy: we use a first lense, whose resolution is not very high but the view is fairly broad, to examine `Constructs` and identify those that seem (roughly) similar; we then use a second lense with a much higher resolution but narrower view to zoom in to each group of similar `Constructs` and find differences (or inconsistences) among the members. Obviously, the first step clustering should be somewhat coarse-grained and highly efficient whereas the second step clustering should be fine-grained and able to accurately detect subtle yet critical inconsistences. Next, we explain these two steps of clustering, respectively.

4.3.1 Functionality Clustering

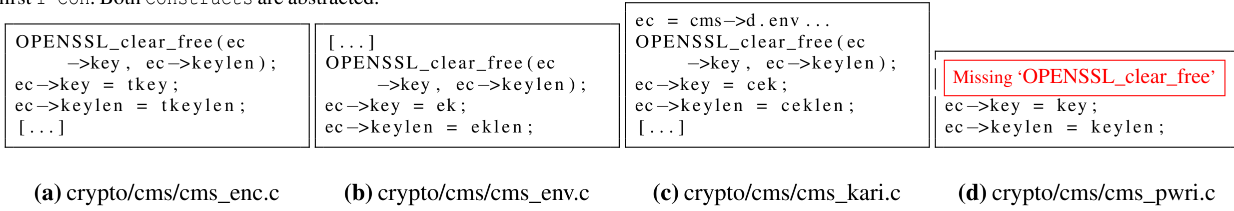
Finding similar constructs is not the same as finding identical ones. Our clustering needs to tolerate mild variations among functionally similar `Constructs`. Otherwise, the first-step clustering may not view a buggy

[2] In static single assignment (SSA) form, existing variables in the original IR are split into versions.



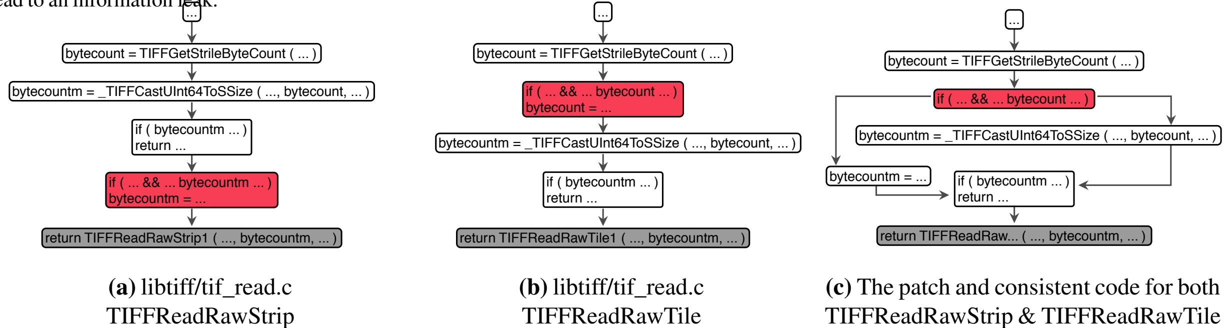
(a) C code example. **(b)** Data Dependence Graph of 'data' Variable. **(c)** Data Dependence Graph of 'data' Variable for the first basic block.

Figure 4: A simple example of an out-of-bound write bug. The first graph shows the `full-Con` for the 'data' variable and the second graph is its first `1-Con`. Both `Constructs` are abstracted.



(a) crypto/cms/cms_enc.c cms_EncryptedContent_init_bio **(b)** crypto/cms/cms_env.c cms_RecipientInfo_ktri_decrypt **(c)** crypto/cms/cms_kari.c CMS_RecipientInfo_kari_decrypt **(d)** crypto/cms/cms_pwri.c cms_RecipientInfo_pwri_crypt

Figure 5: A bug in OpenSSL found by `FICS` based on clustering `1-Cons`. '$ec->key$' has to be cleansed before a new assignment otherwise it might lead to an information leak.



(a) libtiff/tif_read.c TIFFReadRawStrip **(b)** libtiff/tif_read.c TIFFReadRawTile **(c)** The patch and consistent code for both TIFFReadRawStrip & TIFFReadRawTile

Figure 6: An inconsistency in LibTIFF found by `FICS`. The cosine similarity between nodes of the constructs is more than 0.98 while the similarity is very low if both nodes and edges are considered. By considering whole graph similarity as the first-step clustering, this inconsistency would be missed.

`Construct` and a non-buggy or patched `Construct` to be similar, and therefore, fail to provide useful input to the second-step clustering. Variations among similar `Constructs` include missing some nodes, different placements of nodes, etc. For example, Figure 6 shows a detected and fixed bug by `FICS` in LibTIFF. The reason for the inconsistency is that there is a mislocated check in the buggy `Construct` that significantly changes the relations between the nodes. If using a standard graph similarity check, the similarity score between the two `Constructs` (Figure 6a and 6c) can be very low (i.e., the two `Constructs` are deemed significantly different). This is because standard graph similarity checks consider the differences in both nodes and edges of two graphs.

To make the first-step clustering somewhat coarse-grained and tolerant of variations among functionally similar `Constructs`, we design a customized graph similarity scoring scheme. We observed that, if edges are excluded from the similarity comparison, the common types of variations among similar `Constructs` no longer drastically affect the calculation of similarity scores. For instance, the similarity score computed without considering edge difference can be close to 98%. In our scoring scheme (for the first-step clustering only), only node labels in `Constructs` (i.e., abstracted LLVM instructions) are considered and each `Construct` is embedded into a node vector (❺ in Figure 3), where the index is the instruction ID and the value is the number of times the instruction appears in the graph. This embedding shares some resemblance with the bag-of-words technique used in NLP. We call our embedding *bag-of-nodes*, which allows for efficient and variation-tolerant computation of `Construct` similarity. Although edges are omitted in this embedding, nodes usually preserve enough information on the semantics of a `Construct`, and therefore, are sufficient for the purpose of (approximate) functionality clustering.

To quantify the similarity between a pair of `Constructs`, we calculate the cosine similarity (❻ in Figure 3) between their corresponding bag-of-nodes embeddings. We choose cosine similarity for its efficiency and its previ-

ous applications for finding similar code [29]. Specifically, consider the example in Table 1, which shows a stack-based buffer overflow (CWE121) when the check on `data < 10` is missing. The cosine similarity between the buggy and the correct `Constructs` (i.e., `full-Con` with `data` as the root variable) is calculated based on the bag-of-nodes embeddings of the `Constructs`. The computed similarity score is 0.96, which indicates that the two `Constructs` are similar with subtle differences. After computing cosine similarity for each pair of `Constructs`, we feed the pair-wise similarity scores into the clustering algorithm.

The clustering algorithm groups similar `Constructs` in a program based on pair-wise similarity scores. Existing clustering algorithms can be divided into two categories [26]. Those in the first category require as input an exact number of expected clusters (e.g., K-means) whereas those in the second category do not (e.g., DBSCAN and connected-components). Since the number of `Construct` clusters is unknown and may vary significantly across codebases, we choose one algorithm from the second category, namely the connected-component algorithm (❼ in Figure 3). It is less complex and performs much faster than other algorithms such as DBSCAN and Affinity Propagation, based on our tests. This algorithm first constructs a similarity graph based on the previously calculated similarity scores. It then forms clusters from highly connected subgraphs [19].

One important parameter of the clustering algorithm is the similarity threshold, which can be tuned by `FICS` users. Tunning this parameter directly affects the number and sizes of the clusters output by the algorithm. The higher the threshold is set, the more clusters are formed and the smaller those clusters tend to be, and vice versa. Based on our experiments with a subset of the Juliet Test Suite [3], we observed that most buggy and patched `Constructs` usually have similarity scores higher than 0.95 calculated on their bag-of-nodes embeddings. However, we also observed similarity scores as low as 0.7 from some real bugs and their non-buggy counterparts (see §5.2). Although one may be tempted to use a low similarity threshold with the hope of finding more inconsistencies and bugs, this runs the risk of `FICS` reporting too many inconsistencies that are not real bugs. As with many clustering-based systems, a single similarly threshold for `FICS` to perform well on all possible inputs does not exist. But finding a suitable similarly threshold for a codebase does not require much knowledge about `FICS` design or heavy engineering efforts. We consider the value range from 0.8 to 0.95 for the similarity threshold in our evaluation (§5), which can obtain meaningful inconsistencies while keeping the false positives low.

4.3.2 Inconsistency Clustering

`FICS` performs a second-step clustering to group `Constructs` of each cluster generated from the first-step clustering. While our bag-of-nodes embedding is suitable for the coarse-grained clustering, it does not meet the needs of the second step clustering. This is because it does not consider edges in `Constructs` and thus cannot fully capture the structures of `Constructs`. For instance, for a bug caused by a wrong order of operations (CWE666), the nodes of the buggy and non-buggy `Constructs` can be identical (i.e., the same bag-of-nodes embedding), despite the difference in the order of certain nodes. This and other edge-based inconsistencies cannot be captured by our bag-of-nodes embeddings. Therefore, we need a more precise and detailed graph similarity checking scheme.

Graph isomorphism can be used for graph matching. However, it is NP-complete and prohibitively expensive when the number or size of graphs are very big. Other approaches like graph kernels and graph embedding techniques are more efficient. Both the approaches recursively decompose graphs into atomic substructures. Graph kernels define a similarity (aka kernel) function over the substructures [22] whereas embedding techniques use a 'skipgram' model to learn distributed representations [17].

We adopt graph embedding in the second-step clustering because it can learn embeddings automatically while graph kernels require handcrafted substructures. Graph embedding techniques embed either graph substructures (e.g., nodes [10] and paths [31]) or the entire graph [18]. Because our goal in this step is to cluster graphs, not their substructures, we use graph2vec embedding [18], which was recently proposed and can model both local and global similarities among graphs. Based on our experiments, graph2vec achieves similar or better clustering results compared to the other approaches in a much more efficient way.

For each cluster of similar `Constructs`, the similarity between each pair of `Constructs` is calculated based on the graph2vec embeddings (❽ in Figure 3). The similarity scores are then fed into the clustering algorithm (❼ in Figure 3). We use a very high similarity threshold, namely 1 or very close to 1, for the second step clustering, which needs to be sensitive to subtle differences among similar `Constructs`.

4.4 Deviation Analysis and Filtering

Deviation analysis: The output from the two-step clustering contains similar yet inconsistent `Constructs`. However, not all of them are harmful bugs. The deviation analysis (❾ in Figure 3) helps `FICS` users vet the detected inconsistencies in order to quickly identify true bugs.

```c
int data = 10; int i;
int buffer[10] = { 0 };
if (data >= 0 && data < 10 ) {
buffer[data] = 1;
for(i = 0; i < 10; i++)
  printIntLine(buffer[i]);  }
```

Embedding	alloc...	getelem...	icmp sge...	icmp slt...	load...	sext...	br...	call...	store...
Correct	1	1	1	1	4	1	2	1	3
Buggy	1	1	1	0	3	1	1	1	3
Cosine Similarity:	**0.96609** (*i.e.*,>95%)								

Table 1: Computed cosine similarity between the bag-of-nodes embeddings of the correct and the buggy (inconsistent) `Constructs`. Bag-of-nodes embedding in this example is for `full-Con` with *data* as the root variable. The condition `data < 10` is missing in the buggy code, causing the embedding value for the instruction `icmp slt` to be 0. Note that even though only a single statement in C code is missing in this bug, it translates to multiple missing LLVM IR instructions, thus the different values of `load` and `br` in the embeddings.

Inconsistency Type / Deviation	Bug Category
Check `icmp` Node	NULL Pointer Dereference, Undefined Behavior Buffer Errors , Integer Overflow
Memory Handling `*free*`, `*close*` Nodes `*bzero*`, `*clear*` Nodes	Resource Leak, Double Free Information Leak
Type `trunc`, `bitcast` Nodes	Bad Casting
Order Edge	Wrong Order of Operations
Initialization `store`, `memset` Nodes	Double Free, Information Leak

Table 2: List of important types of inconsistencies and deviations that can help detect different types of bugs. Red color refers to LLVM instruction and orange color refers to function call. '*' here means Kleene Star in regular expression.

Most bugs are due to missing or extra code fragments, such as wrong/missing checks (CWE131, CWE190, CWE253, CWE476, CWE475), missing variable initializations (CWE457), missing important calls like `free` or `memset` (CWE200), or redundant calls (CWE415). When expressed in a DDG, these bugs appear as a node deviation from their functionally-similar and correctly-implemented counterparts. For other bugs, such as wrong order of operations (CWE666), each manifests in a DDG as an edge deviation.

We summarize such deviations in Table 2 to guide and facilitate manual analysis and bug triage. While not meant to be complete, this list helps human analysts prioritize the inconsistencies of higher bug potentials. The inconsistencies that contain such deviations are highlighted by our system and then provided to analysts.

Filtering: The deviation analysis allows analysts to focus on high-priority inconsistencies or highly likely bugs. The filtering step, on the other hand, removes the inconsistencies that are redundant or likely false. For example, if a detected inconsistency involves several missing "*if*" conditions, the inconsistency is less likely to be true or a bug (i.e., developers rarely forget to perform multiple different checks in a small chunk of code).

The filtering step uses four empirical rules, which are generic and simple, to reduce or deprioritize false or unimportant inconsistencies. (i) In an inconsistency report, if all the `Constructs` in the *inconsistent clusters*[3] overlap, the report is ignored. This overlap usually happens when a variable propagates to another within a function, which makes their corresponding `Constructs` look similar. The differences among such similar `Constructs` in the same function usually do not represent inconsistencies or bugs. True inconsistencies typically happen across different functions or compilation units. (ii) If an inconsistent cluster contains more than a fixed number (e.g., 2) of deviating nodes (i.e., nodes in Table 2), the inconsistency is de-prioritized because it is unlikely to be a true inconsistency (i.e., a single inconsistency rarely involves many deviations). We note that this rule only applies to deviating nodes, rather than all nodes in a `Construct`, and thus, does not over-filter. For example, this rule is not triggered when more than 10 lines of code difference exist in an inconsistent but only two of them are "*if*" conditions (*icmp* nodes). (iii) If the same inconsistency is found multiple times, we only report it once. This redundancy occurs because the system integrates the reports generated using different granularities and similarity thresholds. (iv) If the number of *inconsistent clusters* is bigger than a threshold (5 in our case), the inconsistencies in these clusters are de-prioritized because the more *inconsistent clusters* are identified, the less likely these clusters represent true inconsistencies (i.e., the clusters are fragmented and not reliable).

5 Evaluation

In this section, we evaluate `FICS` by finding answers to the following questions:

- **Q1**: How effective is our method at detecting bugs (§5.2)?
- **Q2**: Is our method able to find unknown bugs in well-tested large codebases without requiring heavy manual validation efforts (§5.3)?
- **Q3**: What are the non-buggy inconsistencies and whether developers consider them fix-worthy? (§5.4)
- **Q4**: How scalable is our approach (§5.5)?

5.1 Testset & Setup

Before discussing the evaluation results, we explain the codebases/testsets and the setup that we used to perform the experiments. To investigate the effectiveness of

[3] *Inconsistent clusters* refer to the result of the 2nd-step clustering inside a *functionally-similar cluster* produced by the 1st-step clustering.

ML-based systems, a testset with ground-truth labels is needed. Although there are some public datasets [13, 20] suitable for supervised learning approaches, none of them were created based on code inconsistencies.

iBench: Since no testset for evaluating the effectiveness of inconsistency detection exists, we created a testset/benchmark, named `iBench`, based on 22 known bugs in real software. While impossible to find all consistency-induced bugs in all software, our manual search yield 22 reported bugs in five different codebases, including several Linux drivers, OpenSSL, libzip, and mbedtls. The selected bugs span nine distinct categories, as shown in the first two columns of Table 4. Using `iBench`, we performed a controlled experiment to answer Q1. In this experiment (§5.2), we evaluated `FICS` on `iBench` and compared it with the existing approaches.

Five codebases of varying sizes: To answer Q2-Q4, we perform a separate experiment (§5.3), where we evaluated `FICS` on five real codebases in their entirety. This experiment examines if `FICS` can find previously unknown inconsistency-inducted bugs in both small and large codebases. Also, it examines how many of the unknown bugs found by our system can be detected by the existing tools.

The five codebases used in this experiment are popular open-source projects of different kinds and sizes. Some are widely considered as well-tested and high-quality. Table 3 shows the selected codebases, their descriptions, and the numbers of contributors. The sizes of the codebases are shown in Table 5, which vary from small (e.g., LibTIFF with 38 compilation units and 6.9K SLoC) to large (e.g., QEMU with more than 2,000 compilation units and 1.7M SLoC). These codebases also represent a good mix of libraries, applications, and system software.

We compared `FICS` with the related bug detection techniques in both experiments. Although there are more candidates to compare, the three detectors we choose, namely, APIsan [32], Crix [15], and LRSan [24], are inconsistency-related, the most recent, and publicly available. While APIsan and Crix run on the testing codebases without any additional adjustment, we had to slightly modify LRSan. This is because LRSan was originally designed to find bugs in the Linux kernel and it relies on the error codes specific to the kernel to detect security checks. As a simple tweak, we changed LRSan to treat negative return values as error codes.

5.2 Controlled experiment

To measure the performance of `FICS`'s two-step clustering approach and its inconsistency detection, we conducted a controlled experiment on `iBench`.

When using `FICS` to detect inconsistencies and bugs, one can adjust two parameters to achieve the best results: the granularity (or `Construct` size) and the `Construct` similarity threshold for the first-step clustering. Depending on the nature of bugs and codebases, different combinations of these two parameters may result in different detection results.

Name	Commit	#Contrib.	Description
QEMU	7a5853c	1,068	Emulator/Virtualizer
OpenSSL	a75be9f	448	TLS/SSL library
wolfSSL	c26cb53	57	TLS/SSL Library
OpenSSH	c2fa53c	42	SSH Tool
LibTIFF	19f6b70	38	TIFF Library

Table 3: Test codebases sorted by the number of contributors ('Contrib.' column). The 'Commit' column indicates the last commit of the codebase, analyzed by our tool.

Based on our experiments, we found that two particular configurations of `Construct` size, namely `1-Con` and `full-Con`, generally perform well in practice. We evaluated both in our experiment. We used four different similarity thresholds in our evaluation: 80%, 85%, 90%, and 95%. If the threshold is set too low, the similarity clusters may become too bigger, which can in turn cause too many falsely detected similar code snippets (see Appendix A).

Comparison: As shown in Table 4, using `iBench`, `FICS` reported 82 inconsistency cases after filtering 410 reports. We compared `FICS` with the three related bug detectors. `FICS` achieved the highest true positive rate (86%) and significantly outperformed the second best (APIsan at 27%). `FICS` missed one bug in the 'wrong value' category because this bug is caused by a constant value but `FICS` removes literals in the `Construct` abstraction step as a design choice. `FICS` also missed two other bugs, one in 'missing free memory' and one in 'missing return value check'. The former was not correctly clustered with its similar code and the latter was mistakenly filtered out by `FICS`. As for the false positive rate, Crix scored the best at 0%, although it has a very high false negative rate (91%). `FICS` has the second lowest false positive rate. Although `FICS`'s false positive rate is much lower than that of the related works, including APISan and LRSan, the absolute number (76%) may still seem high. We note that it should not severely impact the usefulness of our tool. This is because (i) a report produced by `FICS` does not require heavy manual effort to validate (see §5.3.1), and (ii) the filtering step can be expanded to further reduce false positives.

The focus of Crix and LRSan is only on detecting missing checks. They were not designed to detect other types of bugs or inconsistencies, such as bad casting, memory and information leak, uninitialized variables, etc. APIsan aims to detect API misuses. Three out of four bugs found by APIsan are missing/incorrect checks on function return values. APIsan cannot detect the following types

of bugs that FICS can: (1) no API calls are involved in a bug (e.g., bad casting or an uninitialized variable); (2) the buggy code uses an API but the non-buggy code does not, or the opposite; (3) no majority exists to determine the non-buggy code. Similar to APIsan, Crix uses majority voting to capture an inconsistency. Therefore, neither can detect one-to-one inconsistencies.

Overall, the result shows that FICS is highly effective at detecting inconsistency-induced bugs and its detection is agnostic to bug types. In contrast, the three bug detectors evaluated in this experiment target only specific types of bugs and suffer from either high false positives or low true positives when being used for detecting inconsistency-induced bugs in general.

***off-by-one* errors:** As discussed in §4, for mitigating false positives, FICS ignores literals in the similarity and inconsistency analysis. However, doing so may (mildly) limit FICS's ability to detect certain bugs, such as *off-by-one* errors, which differ from the non-buggy/patched counterparts sometimes only in literals. An off-by-one error occurs when the size of an array (or similar data structure) is miscalculated by one, usually causing a loop to iterate one more/less time than needed.

We conducted an experiment to understand the potential negative impact of forgoing literals in our code abstraction. We randomly selected 11 CVEs and checked if FICS can detect them (see Appendix B). While the root cause for all the bugs is the same (a miscalculated boundary condition), interestingly, developers patched the bugs in two different ways. Among the 11 CVEs, only four of them were fixed by adding or deducting an integer value in the condition. In such cases, even if the similar correct code exists, our approach would miss the bugs. On the other hand, seven out of the 11 bugs were fixed in other ways without changing any literals (*e.g.,* replacing $<$ with $\leq$ or adding a missing check). In these cases, which are more common, the difference between the buggy and non-buggy code does not lie in literals. Therefore, forgoing literals in our abstraction does not quite hurt FICS's bug detection ability while greatly mitigating false positives caused by literals.

5.3 Discovered Unknown Bugs

In a second experiment, we applied FICS on the five open-source software of different kinds and sizes (Table 3) to discover new bugs. For simplicity, we set Construct granularity to 1-Con and full-Con and the similarity threshold to 95%. We chose 95% as the similarity threshold for this experiment because it generates the least amount of bug reports, which need to be manually validated for this evaluation. To further reduce the manual efforts required in this evaluation, we focused on reports in two major inconsistency categories, namely check and call inconsistencies, which cause many bugs [15, 32] in the real world. Table 5 shows the statistics on the codebases used in this experiment, including the numbers of source lines (SLoC), bitcode files (compilation units), functions, and constructs.

	Total bugs	FICS	APIsan	LRSan	Crix
Total Number of reports		82	100	5	2
Memory leak	6	5	2	0	0
Information leak	2	2	0	0	0
Bad casting	2	2	0	0	0
Missing argument check	3	3	0	0	1
Deadlock	1	1	1	0	0
Mislocated check	1	1	0	0	0
Missing return value check	4	3	2	0	1
Uninitialized Variable	2	2	0	0	0
Wrong value	1	0	1	0	0
True bugs	22	19	6	0	2
TP rate		86%	27%	0%	9%
FP rate		76%	94%	100%	0%

Table 4: Bug detection results on iBench: a comparison with three related detectors

Name	SLoC	# Bitcode Files	# Functions DDG	# Construct full-Con	1-Con
QEMU	1.7M	2,120	53,625	207,886	419,982
OpenSSL	517K	690	9,802	32,056	75,787
wolfSSL	396K	44	1,519	8,014	23,029
OpenSSH	93K	228	2,047	11,810	33,031
LibTIFF	69K	84	1,245	9,189	28,537

Table 5: Statistics regarding the analyzed codebases. The codebases have been compiled with default compile options and for Linux platform so it might happen that some C files are not compiled and consequently their corresponding bitcodes cannot be generated.

5.3.1 Result

As shown in Table 6, after the filtering step (§4.4), FICS reported a total of 1,821 code inconsistencies. We manually vetted all of them and found 218 to be true or valid inconsistencies. Among them, 95 are code smells (§5.4) and 121 are potential bugs (verified by ourselves).

The manual vetting effort is not as heavy as required to validate results from many other static analyzers. The ease of manual validation of FICS's reports is largely due to the presence of both the consistent and the inconsistent Constructs and the highlighted differences. Showing this contrast when reporting a bug helps developers quickly determine if the bug is valid or harmful. On average, our testers, having little familiarity with the codebases, took less than two minutes to validate an inconsistency report. They used less than 10 hours to analyze all of the 310 reported inconsistencies in OpenSSL. We expect original developers to take even less time.

So far we have reported 36 of the 121 potential bugs to original developers and received 22 confirmations (the other reports are still pending). The confirmed bugs have been patched either by our pull requests or by the developers themselves based on our reports. Some bugs have more obvious security implications than others.

Name	# Reported inconsistencies Total	Check + Call (Sum)	After Filtering	Valid Cases	Code Smells	Potential Bugs	Confirmed Bugs
QEMU	12,320	3,907 + 3,170 (7,077)	1,206	79	26	53	4
OpenSSL	2,419	1,158 + 347 (1,505)	310	59	24	35	9
wolfSSL	586	296 + 124 (420)	91	23	18	5	3
OpenSSH	1,063	509 + 208 (717)	121	29	18	11	1
LibTIFF	925	390 + 156 (546)	93	28	9	19	5
Total	17,313	6,260 + 4,005 (10,265)	1,821	218	95	121	22

Table 6: `FICS` detected 218 valid inconsistencies, including 121 potential bugs (harmful inconsistencies) and 95 code smells (harmless inconsistencies). Among the potential bugs, 22 have been confirmed and fixed by developers so far. Analyzing each report takes no more than 2 minutes.

5.3.2 Case Studies & Security Impact

Missing Checks: About 70% of the detected bugs are caused by missing checks. This matches the findings reported by other researchers [15] that missing checks are a fairly prevalent class of bugs. Two of the missing checks in OpenSSL and one in wolfSSL may lead to `NULL` dereference. Others have different security consequences such as undefined behaviors, crashes, or malfunctioning. Two of the bugs in wolfSSL, missing checks on the input file size, may cause denial of service when exploited by an attacker using a large file. Interestingly, inspired by our report, the developers added sanity checks to 13 other places in the codebase, resulting a major patch of 250 lines of code.

Memory/Information Leak: Another common type of inconsistencies is call deviations, including missing or wrong use of critical function calls, such as those used for freeing memory. Two main consequences of such bugs are: (1) memory leak if memory is not freed after allocation; and (2) information leak if sensitive information like encryption keys are not cleared in memory after use. 20% of the confirmed bugs, including two in OpenSSL and one in OpenSSH, belong to this category.

Other Bugs: The three remaining bugs (i.e., 10%) are also related to check or call inconsistencies. One of them is a bad casting in a condition check and another one is an uninitialized variable. However, their security impact is not immediately clear. Without an in-depth understanding of the codebases, we were unable to manually confirm if these bugs can directly lead to any security consequence or be exploited by attackers. Nonetheless, in general, bad castings could cause type confusion and in turn integer overflows or logical bugs. Uninitialized variables could lead to information leak or logical errors. The last bug does not have any security implication but it negatively affects the performance: the much heavier `OPENSSL_clear_free` is used when `OPENSSL_free` suffices.

5.3.3 Comparison

In this experiment, we again compare `FICS` with the three related bug detectors, APIsan, LRSan, and Crix. Unlike the previous comparison, which was based on a controlled experiment and focused on true/false positive rate, this comparison aims to show how many of the `FICS`-detected bugs can be caught by the three existing detectors. We used the 22 developer-confirmed bugs as the ground truth and applied the three detectors to the codebases. Table 7 shows the number of bugs reported (#Reports) by each detector as well as how many bugs in the ground truth were detected (#Bugs). APIsan produced more than 18,000 bug reports, which include only four of the 22 confirmed bugs. With much fewer bug reports produced, LRSan and Crix each found only one of the 22 bugs. There are two primary reasons for the three detectors to have much lower detection rates than `FICS`. First, all of them target only a specific class of inconsistencies or bugs, namely missing checks or API misuse. Second, two of them are based on majority voting and they cannot detect one-to-one inconsistencies. This result echos the advantages of `FICS`, in particular, its bug-type-agnostic detection.

	`FICS`		APIsan		LRSan		Crix	
	#Rep	#B	#Rep	#B	#Rep	#B	#Rep	#B
QEMU	1,206	4	5,805	0	129	0	98	0
OpenSSL	310	9	7,874	0	30	0	54	1
wolfSSL	91	3	1,049	1	62	0	62	0
OpenSSH	121	1	2,740	0	0	0	5	0
LibTIFF	93	5	645	3	12	1	3	0

Table 7: Comparison between `FICS`, APIsan, LRSan, and Crix on bug detection capability. `FICS` outperforms its competitors while not reporting too many potential cases. #Rep: Number of reports, #B: Number of bugs.

5.4 Code Smells and False Inconsistencies

We studied the detected inconsistencies that are not true bugs. There are two categories of them: (1) true yet seemingly harmless inconsistencies; and (2) false inconsistencies. The cases in the first category are essentially *code smells*, a term used in the software engineering community [9] to refer to any subtle pattern in code that may indicate or become a problem, broadly defined. We note that, despite the implicit or little security impact, code smells detected by `FICS` are true inconsistencies and still worth fixing (examples discussed shortly). The cases in the second categories are falsely detected inconsistencies.

Table 6 shows that 95 of the reported inconsistencies are code smells and valid inconsistencies. Fixing them can improve code quality. Moreover, such code smells

may help developers find deeper problems in their code. For example, we reported one of the detected code smells, a check inconsistency, to QEMU developers. During the investigation of the report, they found and fixed 6 use-after-free bugs [2].

Redundant code is a common type of code smell detected in our experiment, especially extra sanity checks. The main reason for such inconsistencies is the lack of consensus among developers as to where/when to perform certain checks. For example, some developers check the input parameters inside a function while others perform the same check when they call the function. We reported several cases like this. While some developers removed the redundant checks for performance concerns, others preferred to keep the redundant sanity checks for the peace of mind. Either way, revealing such problems helped developers define a uniform API specification to prevent inconsistencies in API usages.

Another common type of code smell detected is the failure to use existing utility functions. We observed several cases where the inconsistent code performs certain operations by itself instead of calling an existing utility function that does the same. Although such inconsistencies may not have direct security implications, they may give rise to bugs when an update to a utility function cannot propagate to the inline counterparts.

Unlike code smells, false inconsistencies occur when `FICS` mistakenly marks perfectly fine code as inconsistent. For example, when `FICS` learns from several similar `Constructs` that a new value is assigned to a pointer only after the pointer has been freed. `FICS` may detect another similar `Construct` to be inconsistent if it fails to free a pointer before assigning a value to the pointer. However, the "inconsistent" `Construct` in this case assigns a value to an uninitialized (or null) pointer, which should not be freed in advance. `FICS` may not be able to capture such subtle differences in semantics and thus report false inconsistencies. Due to the lack of ground truth, we were unable to determine the exact number of false inconsistencies in this experiment. However, based on our sampling, the rate matches the false positive rate reported in the controlled experiment (§5.2).

5.5 Performance

We run the experiments on a 20-core workstation with 200 GB of RAM. The entire analysis process finishes within five hours for three out of five codebases. The two outliers are OpenSSL and QEMU, taking 12 and 72 hours to analyze, respectively (see Appendix C). In general, a codebase with a larger number of compilation units require longer time for analysis. As for the time spent on individual analysis steps, the second-step clustering is the most time-consuming step, primarily due to the graph embedding generation.

6 Limitations

`FICS` cannot, and is not designed to, detect buggy `Constructs` that do not have functionally similar and non-buggy counterparts in the same codebase. Our approach learns from the codebase itself. If the size of the codebase is too small, the system is less likely to be able to find enough similar `Constructs`and thus inconsistencies and bugs. We note that focusing on large codebases for bug detection is of significant practical value. Moreover, the unique advantage of our approach is the ability to learn from a codebase without prior knowledge about bug patterns or types.

Certain bugs (e.g., one-liners) may be too small to be captured by `FICS` because the smallest `Construct` granularity contains a full basic block and can still be too big for these tiny bugs (i.e., the functional similarity disappears when viewed in large `Constructs`).

Our research prototype currently does not support C++ because the DDG extraction step does not handle C++-specific instructions such as those related to vectors or exceptions. Moreover, our current prototype cannot analyze extremely large codebases (e.g., the Linux kernel). This limitation is due to the very large RAM (>200GB) needed for performing the graph-based code similarity comparison on codebases with more than 16,000 compilation units.

7 Conclusion

In this paper, we presented `FICS`, the first bug-generic, ML-based bug detection system that learns from the to-be-checked codebase and identifies code inconsistencies as bug indicators. Unlike many previous works, our approach does not require external datasets for training nor is limited to certain types of bugs. `FICS` features several novel concepts and techniques, including `Constructs` as the suitable code granularity for similarity/inconsistency comparison, the two-step clustering, and the two graph embedding schemes. These techniques together make `FICS` effective and scalable to large codebases. We applied `FICS` to five popular, well-tested open source projects and found 22 real, previously unknown bugs. All the new bugs have been confirmed or fixed by the developers. We therefore conclude that `FICS` is a practical system and can be directly adopted by developers or testers to find bugs in their code with a minimal amount of manual assistance.

Acknowledgments

The authors would like to thank the anonymous reviewers for their help with the revision of this paper. This project was supported by the National Science Foundation

(Grant#: CNS-1748334) and the Office of Naval Research (Grant#: N00014-18-1-2660). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

Availability

The source code of our system is available here: https://github.com/RiS3-Lab/FICS.

References

[1] https://gitlab.com/libtiff/libtiff/-/merge_requests/96.

[2] https://lists.gnu.org/archive/html/qemu-devel/2020-03/msg07241.html.

[3] Juliet test suite for c/c++. https://samate.nist.gov/SRD/testsuite.php. Accessed: 2018-04-10.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[5] P. Bian, B. Liang, Y. Zhang, C. Yang, W. Shi, and Y. Cai. Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[6] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. Nar-miner: Discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 411–422, New York, NY, USA, 2018. ACM.

[7] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[8] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 480–491, New York, NY, USA, 2016. ACM.

[9] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 175–190, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 855–864, New York, NY, USA, 2016. ACM.

[11] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 55–64, New York, NY, USA, 2007. Association for Computing Machinery.

[12] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2201–2215, New York, NY, USA, 2017. ACM.

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *ArXiv e-prints*, January 2018.

[14] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai. Antminer: Mining more bugs by reducing noise interference. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 333–344, May 2016.

[15] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, Santa Clara, CA, August 2019. USENIX Association.

[16] T. McCarthy, P. Rümmer, and M. Schäf. Bixie: Finding and understanding inconsistent code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 645–648, May 2015.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[18] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, and Yang Liu.

graph2vec: Learning distributed representations of graphs. 2017.

[19] David James Pearce and david. pearce. An improved algorithm for finding the strongly connected components of a directed graph. 2005.

[20] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning. *CoRR*, abs/1807.04320, 2018.

[21] Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. Explaining inconsistent code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 521–531, New York, NY, USA, 2013. ACM.

[22] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, November 2011.

[23] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[24] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1899–1913, New York, NY, USA, 2018. ACM.

[25] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[26] Rui Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, May 2005.

[27] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 363–376, New York, NY, USA, 2017. ACM.

[28] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.

[29] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 359–368, New York, NY, USA, 2012. ACM.

[30] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 499–510, New York, NY, USA, 2013. ACM.

[31] Pinar Yanardag and S.V.N. Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1365–1374, New York, NY, USA, 2015. ACM.

[32] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing API usages through semantic cross-checking. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 363–378, Austin, TX, 2016. USENIX Association.

Appendix A Threshold Breakdown

Figure 7 shows the number of inconsistency reports and the number of confirmed bugs produced by `FICS` under each configuration combination. Based on this result, the higher the similarity threshold is set, the fewer similar `Constructs` are detected and fewer inconsistencies reported, except for the threshold of 80%. This outlier is caused by the the filtering step, which removed many large and false clusters. However, if the threshold is set too high, it is possible that few similar `Constructs` can be detected and thus some true inconsistencies and bugs may be missed. In practice, we combine the results under the four thresholds and remove the duplicates during the filtering step. As for the `Construct` size/granularity configurations, the figure suggests that `full-Con` performs better than `1-Con` in terms of bugs detected. This result is expected because `1-Con` only captures the data flow of a variable within a single basic block. That said, `1-Con` granularity still helps uncover some bugs that `full-Con` cannot, such as those manifest only in a single basic block.

Appendix B Off-by-one CVEs

Table 8 presents the 11 off-by-one CVEs in *open source C* codebases. Seven out of the 11 bugs have been patched without changing any integer literals and the rest needs literal adjustments. This explains that eliminating integer literals in the abstraction step would not harm our detection capability in the majority of cases.

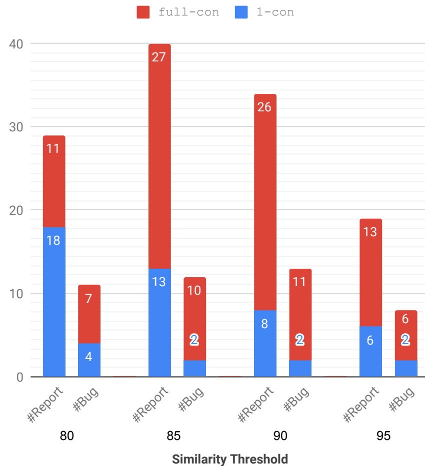


Figure 7: Breakdown of the number of reports as well as the number of bugs.

#	CVE	Patch
		Detection is **NOT** affected by removing integer literals
1	CVE-2020-7044	Replace $<$ with $\leq$ in: $if(in_ptr->max_entry<in_type)$
2	CVE-2018-7329	Replace $\leq$ with $<$ in: $for(i=1;i<=item_count;i++)$
3	CVE-2014-9029	Replace $>$ with $\geq$ in: $if(JAS_CAST(int,coc->compno)>dec->numcomps)$
4	CVE-2014-7937	Fix variable name comparison in: $if(i<vr->ptns_to_read)$
5	CVE-2013-7108	Remove the following redundant statement: $x++$
6	CVE-2011-5244	Add a missing check ($\&\&idx<MAX_NAME$) in: $while(ch!=EOF\&\&ch!=lineterm)$
7	CVE-2007-4091	Add the following missing check: $if(l>=sizeof(fname))$
		Detection is affected by removing integer literals
1	CVE-2019-13306	Change 1 to 2 in: $if((q-pixels+extent+1)>=sizeof(pixels))$
2	CVE-2016-10145	Copy $MaxTextExtent-1$ size in: $strncpy(clone_info->magick,magic_info->name,MaxTextExtent);$
3	CVE-2014-2386	Compare with $MAX_INPUT_BUFFER-1$ in: $if(strlen(getenv("QUERY_STRING"))>MAX_INPUT_BUFFER)$
4	CVE-2006-7221	Remove 1 in: $entry->d_name[MAXNAMLEN+1]='\backslash 0'$

Table 8: List of the 11 analyzed off-by-one CVEs.

Appendix C Performance Overhead

Figure 8 shows the time FICS spent at each step in the analysis pipeline when analyzing the codebases mentioned in §5.1.

We compared FICS, in terms of the running time, with APIsan, LRSan, and Crix (see Table 9). APIsan is more computationally expensive than FICS partly because of its use of symbolic execution. Crix and LRSan are pretty fast and finish their analysis in a few seconds to minutes. This is because their analysis is confined to only missing

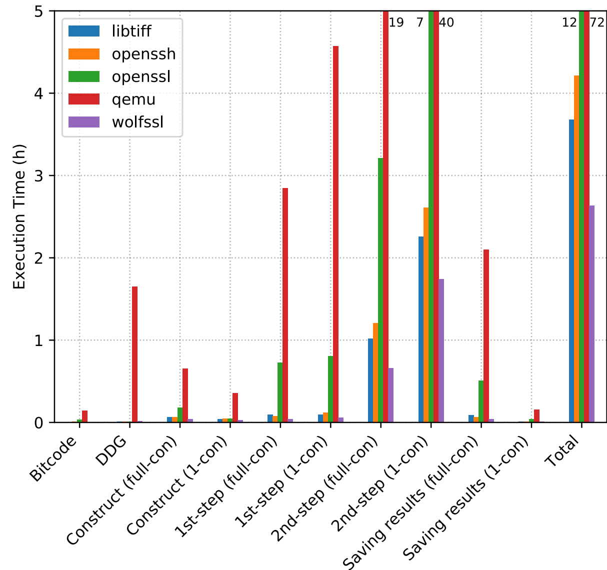


Figure 8: Execution time for different steps of FICS. The most time-consuming step is the 2nd-step clustering.

checks on a limited set of critical variables. Although FICS takes much longer time to analyze a codebase than LRSan and Crix, FICS can detect a far broader range of bugs and code inconsistences. It is not limited to just missing checks on a small set of selected variables as LRSan and Crix are. Furthermore, for an in-depth static analyzer like FICS, spending several or tens of hours on a large codebase is normal and considered acceptable in practice.

It is also worth noting that FICS only needs to perform a full-round of analysis on a codebase once. Its inconsistency detection is by nature incremental: after having analyzed a codebase, FICS can be applied to newly added or changed code without re-analyzing the entire codebase. Furthermore, each step in FICS's analysis pipeline is multi-threaded, which allows for further performance improvement by increasing parallelism.

	FICS	APIsan	LRSan	Crix
QEMU	72 (h)	96 (h)	6 (m)	3 (m)
OpenSSL	12 (h)	92 (h)	34 (s)	28 (s)
wolfSSL	2 (h)	7 (h)	30 (s)	27 (s)
OpenSSH	4 (h)	44 (h)	12 (s)	14 (s)
LibTIFF	3 (h)	18 (h)	1 (m)	23 (s)

Table 9: Performance comparison with APIsan, LRSan, and Crix.