Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs

Qingyang Zhou
zhou1615@umn.edu
University of Minnesota

Qiushi Wu
wu000273@umn.edu
University of Minnesota

Dinghao Liu
dinghao.liu@zju.edu.cn
Zhejiang University

Shouling Ji
sji@zju.edu.cn
Zhejiang University

Kangjie Lu
kjlu@umn.edu
University of Minnesota

Abstract

Security bugs like memory errors are constantly introduced to software programs, and recent years have witnessed an increasing number of reported security bugs. Traditional detection approaches are mainly *specification-based*—detecting violations against a specified rule as security bugs. This often does not work well in practice because specifications are difficult to specify and generalize, leaving complicated and new types of bugs undetected. Recent research thus leans toward *deviation-based* detection which finds a substantial number of similar cases and detects deviating cases as potential bugs. This, however, suffers from two other problems. First, it requires enough similar cases to find deviations and thus cannot work for custom code that does not have similar cases. Second, code-similarity analysis is probabilistic and challenging, so the detection can be unreliable. Sometimes, similar cases can normally have deviating behaviors under different contexts.

In this paper, we propose a novel approach for detecting security bugs based on a new concept called Non-Distinguishable Inconsistencies (NDI). The insight is that if two code paths in a function exhibit inconsistent security states (such as being freed or initialized) that are *non-distinguishable from the external*, such as the callers, there is no way to recover from the inconsistency from the external, which results in a bug. Such an approach has several strengths. First, it is specification-free and thus can support complicated and new types of bugs. Second, it does not require similar cases and by its nature is deterministic. Third, the analysis is practical by minimizing complicated and lengthy data-flow analysis. We implemented NDI and applied it to well-tested programs, including the OpenSSL library, the FreeBSD kernel, the Apache httpd server, and the PHP interpreter. The results show that NDI works for both large and small programs, and it effectively found 51 new bugs, most of which are otherwise missed by the state-of-the-art detection tools.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560661

CCS Concepts

• **Security and privacy → Systems security**; **Software and application security**.

Keywords

Deterministic Bug Detection; Static Analysis; Non-Distinguishable Inconsistencies

ACM Reference Format:
Qingyang Zhou, Qiushi Wu, Dinghao Liu, Shouling Ji, and Kangjie Lu. 2022. Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560661

1 Introduction

Security bugs have been a major source of cyber attacks. According to Common Weakness Enumeration (CWE), common security bugs include out-of-bound access, improper input validation, NULL dereference, etc. Researchers and security professionals have shown how to exploit such security bugs to take control of a system, leak information, and cause denial-of-service. The causes of security bugs are typically incorrect security-related operations, such as missing a bound check, pointer nullification, or memory release.

Bug detection is a well-explored topic and is still a popular solution to mitigating the threats from security bugs. A first wave of detection approaches is based on specifications. Developers first provide specifications for secure code, and then analyze the code to detect violations against the specifications as potential bugs [6, 8–10, 12, 28, 30, 35, 37]. Well-known problems with such an approach are that specifications are hard to obtain, given the complexity of code, and that understanding and checking the code against specifications can be hard. As a result, existing specifications only cover limited types of bugs with clear patterns.

Recent research has shifted to cross-checking [1, 13, 14, 20, 22, 43] for bug detection. Its idea is to first collect similar code snippets and then identify deviating cases as potential bugs. This is intuitive because in general most of the similar code snippets should be correct. It has helped detect many bugs of various types. An advantage of cross-checking is that it can be agnostic to security operations and does not require the understanding of complicated code. However, the cross-checking approach works based on two assumptions: (1) we could find enough similar code snippets and (2) the similar code snippets should have similar security operations.

In practice, both assumptions may not hold. The first assumption can be invalid due to custom code or inaccurate similarity analysis. Custom code is common in programs that may not have similar code snippets. For example, while there are only a few standard allocation functions (e.g., `kmalloc`), K-MeLD [5] finds more than 800 custom allocation functions in the Linux kernel. On the other hand, code-similarity analysis [11, 15, 16, 33, 40, 42] is considered a hard problem. Existing techniques mainly focus on context-insensitive syntactical or structural similarity instead of code semantics and logic, resulting in inaccurate similarity analysis. The second assumption may also be invalid, as similar code snippets may normally have divergent operations under different contexts. As a result, cross-checking fails in cases lacking similar code snippets, and its detection results are probabilistic and can be unreliable.

In this paper, we present a deterministic oracle for detecting security bugs, namely Non-Distinguishable Inconsistencies (NDI). NDI is based on a fact that if two code paths (no matter if they are similar or not) in a function has an inconsistent security state, e.g., one frees a pointer, but the other does not, the inconsistency must be recovered in the callers, and to recover from the inconsistency, the callers must be able to distinguish which path of the two has been taken, which requires a distinguisher variable (e.g., different return values). In cases where such distinguisher variables do not exist or are overwritten, the callers will never be able to recover from the inconsistent security state, so we can reliably conclude that this constitutes a bug. Though NDI is deterministic theoretically, false positives still exist in practice, because of certain implementation limitations, which will be discussed in 6.6.

NDI has multiple advantages over existing bug-detection approaches. The most important one is that its detection can be deterministic in principle. As long as the inconsistent paths are non-distinguishable from callers, the inconsistency cannot be recovered, leading to a bug. Second, the detection tends to be *localized*. The analysis scope is typically limited to the code paths within a function and a few callers. It does not require the analysis of lengthy or complicated data flows. Third, the detection does not require specifications or similarity analysis, and thus is able to generalize. For example, unlike cross-checking, NDI can also work well on small projects that contain much less similar paths.

We realize the NDI approach through three phases. First, given a function, we enumerate any two code paths, which are likely not similar, and identify if they have an inconsistent security state or operation such as memory free. We define the variable involved in the security state or operation as *critical variable*. Second, for a path pair with an inconsistency, we comprehensively identify any variable that is able to distinguish the paths. Such a variable is typically used in an `if` statement or a `return` statement. If there is no distinguisher, we already know it is a potential bug. Otherwise, we will move to the third phase—analyzing the callers of the given function to see if they ever recover from the inconsistency using distinguishers before the critical variable is used. If not, we report it as a potential bug. In addition, if the distinguishers are overwritten without recovery, we also conclude it is a potential bug.

When achieving NDI, we identify two technical challenges. The first is identifying the distinguisher—any variable that is visible outside the path pair and can distinguish which path of a pair has been taken. That is, we should guarantee that a distinguisher variable will always contain differing values when the two paths of a pair are executed. Also, the distinguisher variables can take various forms such as arguments, return values, global variables, or aliases of them. The identification should be comprehensive to not miss distinguishers, which is essential to minimize false positives in the bug-detection phase. Second, inherently the analysis of NDI is both inter-procedural and path-based, which is susceptible to path explosion, especially when checking if the callers ever recover the inconsistencies. To ensure scalability, specific techniques are required to mitigate the path explosion.

To address the first challenge, we propose a new technique to precisely identify distinguishers. In particular, we symbolically execute the two paths of a pair while normalizing the symbols based on the alias analysis. By modeling the "differing" as a constraint and leveraging the conservativeness of under-constrained symbolic execution, we identify symbolized variables as distinguishers that must contain differing values along the two paths. As for the second challenge, we develop a technique to limit our analysis scope. The technique leverages function-level slicing to minimize the scope for precisely checking the recovery of inconsistencies in the callers. With the techniques, as confirmed in the evaluation, NDI can detect security bugs accurately and scalably in real-world projects.

We implement NDI based on LLVM and apply it to both large and small projects including the `OpenSSL` library, the `FreeBSD` kernel, the `httpd` server, and the `PHP` interpreter. The experimental results show that NDI detects many new bugs that were missed by previous tools within 5 hours. Until the paper submission, NDI has detected 51 new bugs in the projects; 46 have been confirmed and 25 have been fixed by the maintainers. NDI is open-sourced [1] and can be used to find more bugs in other projects. In summary, we make the following contributions.

- **A new, deterministic approach for detecting security bugs.** We propose *non-distinguishable inconsistencies as a deterministic detection oracle* based on an observation that inconsistent security operations cannot be recovered when there is no distinguisher, and will lead to a bug. Compared to traditional approaches, the detection does not require specifications, code-similarity analysis, or complicated data-flow analysis. The approach can also generally work for programs of various scales and with custom code.
- **New techniques for detecting non-distinguishable inconsistencies.** We propose new techniques to overcome challenges in achieving NDI. The first technique relies on a new under-constrained symbolic execution to accurately identify all distinguishers. The second technique mitigates path explosion by performing a two-stage analysis: scalable scope analysis and precise recovery analysis. With the techniques, NDI can accurately and scalably detect security bugs in real-world large projects.
- **New bugs on well-tested critical projects.** We found 51 new bugs in well-tested real-world projects of different scales, including `OpenSSL` library, `FreeBSD` kernel, `httpd` server and `PHP` interpreter. Most of the bugs are security-critical as they may lead to memory corruption or crashes. We have reported these bugs; 46 have been confirmed by the maintainers.

[1] https://github.com/umnsec/ndi

2 Motivation

Decades ago, researchers detected violations against specifications as bugs. Given the challenges in specifying rules and understanding the code semantics, recent research favored cross-checking which collects similar code snippets and detects deviations among them as potential bugs. Such an approach does not require specifications or an understanding of complicated code logic. In this work, we identify some fundamental limitations with existing approaches that motivate us to propose NDI. In this section, we present the limitations with examples and explain why NDI can address them.

2.1 A Motivating Example

Figure 1 shows a wild-pointer-dereference bug in the `OpenSSL` library captured by NDI. In this case, function `evp_pkey_get_legacy` has inconsistent return states: returning NULL on failure but a non-NULL pointer on normal execution. The caller chain of `evp_pkey_get_legacy` is: lines (18)–(25). The uses of the return value of `evp_pkey_get_legacy` are: lines 31, 36, and 42. Interestingly, `&dh->params` is not a dereference of `dh`, as compilers will optimize the "dereference" away to directly obtain the address (base + 0x8) of the variable `dh->params`. Therefore, the actual critical use of the return value is on line 42. While the return value itself and `pk->lock` are distinguishers, they are never checked against before line 42, which constitutes a wild-pointer-dereference bug or even arbitrary read. If the target system marks the zero page inaccessible, the code will crash on line 42; if the target system allows access on zero page, such as in some embedded systems without MMU, the bug can turn into an arbitrary read when attackers control the memory at 0x8.

```
1  void *evp_pkey_get_legacy(EVP_PKEY *pk) {
2      void *ret = NULL;
3      ret = pk->legacy_cache_pkey.ptr;
4
5      if (!CRYPTO_THREAD_unlock(pk->lock))
6          return NULL;
7
8      if (ret != NULL)
9          return ret;
10     ...
11 }
12
13 DH *evp_pkey_get0_DH_int(const EVP_PKEY *pkey) {
14     if (...) {
15         ERR_raise(...);
16         return NULL;
17     }
18     return evp_pkey_get_legacy((EVP_PKEY *)pkey);
19 }
20
21 static int dh_pkey_ctrl(EVP_PKEY *pkey, int op, ...) {
22     switch (op) {
23     case ASN1_PKEY_CTRL_SET1_TLS_ENCPT:
24         ...
25         return ossl_dh_buf2key(evp_pkey_get0_DH_int(pkey), ...);
26     }
27 }
28
29 int ossl_dh_buf2key(DH *dh, ...) {
30     ...
31     DH_get0_pqg(dh, ...);
32     ....
33 }
34
35 void DH_get0_pqg(const DH *dh, ...) {
36     ossl_ffc_params_get0_pqg(&dh->params, ...);
37 }
38
39 void ossl_ffc_params_get0_pqg(const FFC_PARAMS *d,
40                               const BIGNUM **p, ...) {
41     if (p != NULL)
42         *p = d->p;
43 }
```

Figure 1: A wild-pointer dereference bug found by NDI. It can be an arbitrary read of the zero page is attacker controllable.

2.2 Why Existing Approaches Fail

Here, we explain why existing approaches such as data-flow analysis and cross-checking would fail.

Complicated specification and data flows. While in this bug it could be a traditional wild-pointer dereference, specifications in general are complicated. From the example, we can see that the analysis must be inter-procedural and often inter-module, starting from the source to uses. The original code of the example actually involves pointer aliases, lengthy and complicated data flows. Existing data-flow analysis is limited in handling such lengthy data flows because of aliasing and path explosion.

Hard to find similar cases. If one instead turns to cross-checking, finding similar code snippets is required. Traditional similarity analysis may not find enough similar cases. For instance, the wrapper function `evp_pkey_get0_DH_int` is only called three times in the whole `OpenSSL` project, and only one callsite is followed by a security check of the return value `ret`. That is, the example is actually violating cross-checking assumption, because the "majority" of cases are incorrect in this example.

Recent research attempted to improve the similarity analysis by focusing on only relevant paths and data flows. For example, FICS [1] uses intra-procedural data dependence graph to represent relevant code in a fine-grained way. In our example, this bug is inter-procedural and involves many aliases, so FICS will miss this bug. IPPO [17] tries to collect similar path pairs and detects inconsistencies as potential bugs. To reduce false reports, IPPO has aggressive rules for selecting similar paths. For example, the paths must start at the same block and end at the same block, and have the same state (i.e., both are on the error path or non-error path). RID [21] even assumes that the two paths of a pair must have the same return value and argument value. Such restrictive rules will clearly exclude the bug because at least the paths have differing states and return values.

In fact, code-similarity analysis is a hard problem [11, 15, 16, 33, 40, 42]. It is still difficult to achieve high precision. This results in that bug detection based on similarity is inherently probabilistic. More importantly, in small or custom projects, similar code snippets may not exist at all.

Deviations can be legitimate or do not exist. Even if one can find similar code snippets, deviations may not constitute real bugs. The reason is that how the code should perform security operations also depends on its contexts. For example, missing a bound check inside a function may not be a bug if its callers all have enforced the check; also, missing a free may not be considered a bug if it is in a terminating function; Interestingly, bugs might not be the minority cases. In this example, `evp_pkey_get0_DH_int` has in total three uses; it is incorrectly used twice but correctly used only once.

2.3 Why NDI Can Detect the Bugs

The limitations with existing detection approaches motivate us to propose a new approach—non-distinguishable inconsistency. If an inconsistent state between two paths cannot be recovered after the two paths merge and the involved critical variable is used,

it must be a bug. Unlike existing approaches like FICS [1] and IPPO [17], NDI does not require similarity analysis. NDI requires only two conditions. (1) There are two paths in the function that have inconsistent security operations or states, i.e., returning NULL in our example. (2) The critical variable involved in the security operations or states are visible after the two paths merge.

Both conditions are easy to form in functions with branches (e.g., error checks). In the example, the checks at line 5 and line 8 of Figure 1 generate different path pairs, and clearly, the critical variable `ret` is visible after the two paths merge, so NDI is applicable to the case. Although the function `evp_pkey_get_legacy` has distinguishing variables, none of them is checked against; as a result, the inconsistencies will never have a chance to be recovered. Therefore, NDI flags them as bugs.

3 Definitions and Overview

3.1 Definitions

Code path and path pair. We define an intra-procedural control flow within a function as a code path. The code path does not necessarily start from the function prologue or end at the function epilogue. We then define *path pair* as two code paths within the function that start from a *diverging point* (e.g., an `if` statement) and end at a *merging point* (e.g., a `return` statement). For example, in Figure 1, path-1 with lines ⑤-⑥ and path-2 with lines ⑤-⑧-⑨ form a path pair.

Inconsistency. We define *inconsistency* as a divergent security state in a path pair. In this project, as examples, we focus on the common security states, including being NULL, freed, and initialized. In the above path pair, the inconsistency is that the return variable `ret` in path-1 is being NULL while in path-2 is being non-NULL.

Critical variable. We define *critical variable* as the one involved in the inconsistent security state or targeted by the security operation that results in the inconsistency. In this example, it is the return value of `evp_pkey_get_legacy`.

Distinguisher. We define a variable as a distinguisher if it *must* hold differing values when the two paths are executed and is visible after the two paths merge, e.g., after the function returns. After the two paths merge, a distinguisher can be used to tell which path has been taken. For the above pair, the distinguishers are the return value `ret` and `pk->lock` because they can distinguish the paths.

Recovery of inconsistency. Before the critical variable is used, e.g., a pointer is dereferenced, its inconsistent state must be consistentized, which we define as *recovery of inconsistency*. For this particular case, to be consistent, `ret` should be nullified for path-2 or assigned with a non-NULL pointer for path-1.

Non-distinguishable inconsistency—the bug oracle. When there is no distinguisher at all, or distinguishers are all overwritten before the recovery, the inconsistency will become non-distinguishable; when the critical variable is used, the non-distinguishable inconsistency becomes a deterministic oracle for a security bug. In addition, if distinguishers have never been used for recovery when the critical variable is used, we can also conclude this is a bug.

3.2 Overview

The goal of NDI is to identify non-distinguishable inconsistencies as potential bugs in projects of different scales. Given the source code of a project, NDI will go through three phases and automatically report bugs as the output. Figure 2 shows an overview of NDI. In the following, we introduce each phase.

In the first phase, NDI identifies inconsistent path pairs. Given a function, NDI first collects path pairs, as defined in §3.1. Each pair has a diverging point and a merging point. The second step is to check if a path pair has an inconsistent security state. This state can be identified by conditional statements (e.g., `if` statement) or security operations (e.g., memory free). The reason is that a conditional statement often separates a state (e.g., being NULL or not), and a security operation often changes a state. After that, we can quickly identify the critical variable which is used in the conditional statement or is targeted by the security operation. Last but not least, we also need to make sure that the critical variable is visible after the two paths merge (i.e., it is not defined or destroyed within the path pair); otherwise, the inconsistent security state would not impact the following code at all, which is beyond the scope of NDI. By going through these steps, NDI will finally report the *inconsistent path pairs*—path pairs with externally visible inconsistent states.

In the second phase, NDI identifies distinguishers for inconsistent path pairs. This is a challenging and important phase. The identification should be comprehensive but also precise, which is a prerequisite for accurate bug detection. Intuitively, any variable that is visible after the merging point and can distinguish the paths should be considered a distinguisher. To ensure the comprehensiveness and precision, we propose to use a custom under-constrained symbolic execution together with a path-based alias analysis. Given a path pair, NDI first performs our own analysis (see §4.2) to normalize variables involved in the path pair. After that, NDI performs an under-constrained symbolic execution for both paths *separately*, which generates constraints for variables in the path pair. For each variable that is visible after the paths merge, NDI aims to determine if it *must* contain different values after the merging point. To this end, we collect its constraints for both paths and construct an artificial constraint for that they are the same. All these constraints are then sent to a solver; if the result is unsatisfiable, we know that the two paths must result in differing values for the two paths, and we identify the variable as a distinguisher.

In the third phase, NDI detects non-distinguishable inconsistencies as potential bugs. Clearly, if there is no distinguisher at all, NDI can quickly report the case as a potential bug. However, in most cases, there are distinguishers, and NDI needs to analyze the code following the merging point to determine if the distinguishers are used to recover the inconsistency. There are two scenarios that NDI will report as bugs. First, the distinguishers are not used for recovery before the critical variable is used. Second, the distinguishers are overwritten before recovery. A challenge of handling the two scenarios is that NDI has to perform an inter-procedural and path-based analysis, which may suffer from path explosion. To mitigate the problem, we perform a two-staged analysis: a coarse-grained function-level program slicing to limit the analysis scope and a fine-grained recovery analysis to detect bugs. We will discuss the

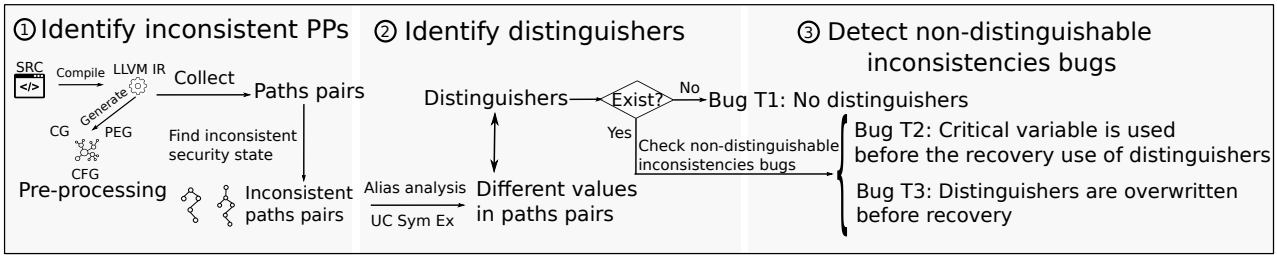

Figure 2: Overview of NDI-based detection. PPs = path pairs.

two-staged analysis in §4.3. In the end, NDI generates bug reports and we manually confirm them.

4 A Bug-Detection System Based on NDI

In this section, we present the design of NDI. First, we list the main technical challenges below.

C1. The first challenge is to identify path pairs with inconsistent security states that are visible to the external.

C2. The second challenge is to identify distinguishers of the inconsistent path pairs precisely and thoroughly.

C3. The third challenge is to precisely and efficiently detect non-distinguishable and unrecovered inconsistencies.

In the following subsections, we present our solutions to these corresponding challenges.

4.1 Identification of Inconsistent Path Pairs

The very first task of NDI is to identify inconsistent path pairs which have inconsistent security states. Given a function, NDI traverses the control-flow graph to collect individual code paths. Note that this step is intra-procedural, as we will not enter into callees. A code path is represented as a list of basic blocks. Then, NDI enumerates the paths to find potential path pairs by checking against the first condition—starting from a diverging point and ending at a merging point. The diverging point can be either the function prologue or a branch statement (typically at the end of a basic block), and a merging point can be either the function epilogue or the targets of a branch. By going through the control flow, we can easily collect such pairs. A more complicated part is to determine that (1) a pair has an inconsistent security state, and (2) the inconsistent security state is visible after the two paths merge.

```
1  dav_error * dav_fs_dir_file_name(const dav_resource *resource,
2                                   const char **dirpath_p,
3                                   const char **fname_p) {
4      dav_resource_private *ctx = resource->info;
5      char *dirpath = ap_make_dirstr_parent(...);
6      if (...) {
7          if (dirpath[dirlen - 1] == '/') {
8              dirpath[dirlen - 1] = '\0';
9          }
10     }
11     if (rv == APR_SUCCESS || rv == APR_ERELATIVE) {
12         *dirpath_p = dirpath;
13         if (fname_p != NULL)
14             *fname_p = ctx->pathname + dirlen;
15     }
16     else {
17         return dav_new_error(...);
18     }
19     return NULL;
20 }
```

Figure 3: Example function in `httpd` with inconsistency

Identifying security states and critical variables. A security state is a program state that is sensitive to security, such as being NULL, being initialized, or being freed. It is typically represented with the value of a variable. The state is often manipulated through a security operation [17, 36], such as memory release, nullification, initialization, etc. A variable that represents the state or is targeted by a security operation is considered a *critical variable*. Following this definition, we can identify security states and critical variables through security operations and the values (e.g., NULL) representing states. For example, Figure 3 shows a function in `httpd` server, which includes the initialization security operation against variables `dirpath` (line 8) and `fname_p` (line 14). Therefore, we identify `dirpath` and `fname_p` as the critical variables. Consequently, their security states become "initialized" after the operations.

Analyzing the visibility of security states. For an inconsistent security state to take effect (leading to a bug), the critical variable must be visible after the two paths merge, so that it can be used and trigger a security bug. Intuitively, to check the visibility, we can perform a backward data-flow analysis to identify the definition (e.g., a definition of an integer or allocation of a buffer) of the critical variable, and make sure it is before the diverging point. However, in NDI, we have a much easier solution, which is to simply check that the critical variable is not defined within the path pair, which suffices to ensure the external visibility.

Precise path-based alias analysis for critical variables. It is not uncommon that the two paths may perform security operations against aliases (with different names) of the same pointer. Precisely identifying the aliases is essential to reduce false positives in inconsistent security states. Existing alias analysis does not work well. While the conservative `MayAlias` results contain overwhelming false positives, the precise `MustAlias` results miss too many true positives. Therefore, we need a new alias analysis that is precise, efficient, and also very comprehensive.

Fortunately, we observe that in NDI, since we only target two paths that start from the same diverging point and end at the same merging point, an intra-procedural path-based and field-sensitive data-flow analysis would be efficient and can precisely resolve the aliases, as such an analysis can precisely keep track of the propagation of pointers in the granularity of memory fields. Note that if a path pair does not start from the function prologue, we will also apply our analysis to the paths from the function prologue to the start of the path pair to capture aliases generated along the paths. During this analysis, if we encounter a function call, we will not track into the callee but use the `MustAlias` results from the existing alias analysis to balance the accuracy and efficiency.

Identifying inconsistent path pairs. With the alias results, we then propose a method to identify inconsistent path pairs. Such pairs *must* have inconsistent security states. Inconsistent security states may be generated in different ways, so we have two criteria to determine inconsistent security states. First, if a security operation (such as initialization or memory release) against a specific variable appears in only one path but not the other, we identify an inconsistent state, and the path pair is inconsistent. Second, since a security state may not be generated through a security operation, but represented by values (e.g., NULL vs. non-NULL), we also perform a value-based analysis. In the current implementation, we focus on NULL pointers. This way, we identify path pairs with inconsistent security states.

4.2 Identification of Distinguishers

One of the key challenges of NDI is identifying distinguishers—variables that tell which path of a pair has been taken, and the variables should be visible outside the pair. To the best of our knowledge, this has not been studied before. There are often many variables involved in a pair, and most of them may or may not be differing after the two paths merge. Determining which variables must be differing is a challenge. Moreover, the analysis should be precise and comprehensive, which is essential to minimize false positives and false negatives in bug detection.

We reason that identifying distinguishers is to determine whether a variable (that is visible outside the path pair) *must* result in differing values at the end of the path pair. Therefore, if a variable *may* result in the same value, it cannot be identified as a distinguisher, as in this case the callers will still not be able to distinguish the paths. Based on the reasoning, we propose to use a tailored *under-constrained symbolic execution* to reliably identify distinguishers. The under-constrained symbolic execution will not only help determine the possible values of a variable, but also exclude infeasible paths to reduce false positives. Also, it will not suffer from path explosion because it only targets two intra-procedural paths.

Under-constrained symbolic comparison with extension. The idea is that given a path pair and a variable, we can symbolize the variable into two different symbols (e.g., V_1 and V_2) for the two paths and symbolically execute the two paths separately to collect two sets of constraints. By adding an artificial constraint that $V_1 == V_2$ and sending all these constraints about the variable to a solver [23], we can get a result which is either `SAT` or `UNSAT`. If it is `UNSAT`, we can confidently conclude that V_1 always differs from V_2, and the variable is a distinguisher. The confidence is based on the conservativeness of under-constrained symbolic execution. While this is intuitive, we still need to address several problems.

Normalizing symbols by extending path pairs. Variables in question must be translated into symbols before symbolic execution for a path pair. Simply translating every variable with a different name on a path to a new symbol is not precise because memory can be pointed to by aliases with different names, and variables with different names may also have the same values. To address this challenge, we propose the path-pair extension method. This method will extend both paths in the inconsistent path pair from the diverging point to the function prologue and generate new path pairs, which we refer to as *extended path pairs*. The paths in one extended path pair will share the same path from the prologue to the diverging point, and we refer to it as a *pre-path* as this path is executed before the inconsistent path pair. Note that one inconsistent path pair can generate more than one extended path pairs, as there may exist more than one pre-paths before the diverging point. By starting the symbolic execution from the function prologue, the same variables with different names can be recognized, and the translation from variables to symbols becomes definitive. On the other hand, we can also collect more constraints for variables. As a result, the identification of distinguishers will become more precise.

Take an inconsistent path pair in Figure 3 as an example. We have a path pair: path-1 ⑪-⑫-⑬-⑭-⑲ and path-2 ⑪-⑯-⑰. From the prologue to the diverging point ⑪, the function `dav_fs_dir_file_name` has three pre-paths: ⑭-⑮-⑯-⑪, ⑭-⑮-⑯-⑰-⑪ and ⑭-⑮-⑯-⑰-⑱-⑪. By composing the pre-paths with the inconsistent path pair, we will generate three extended path pairs respectively.

Note that when identifying the distinguishers of the inconsistent path pair with multiple extended path pairs, our policy is that as long as a variable is a distinguisher for one extended path pair, we deem the variable as a distinguisher. That is, the variable does not have to be a distinguisher for all extended path pairs.

Variable selection and symbolization. Only variables that can be potential distinguishers should be symbolized, so we need selection criteria. To become a distinguisher, a variable should satisfy the following two conditions. First, the variable must be defined before the merging point, i.e., the variable is defined on the pre-path or passed in as a global or argument. Second, the variable must be used in the inconsistent path pair (including the diverging point) to generate differing values. After selecting the candidate distinguisher variables, for each of them, we will symbolized it into two different symbols on the two paths of a pair and perform the symbolic comparison.

Constructing and solving constraints. To determine if a variable, say `V`, must contain differing values after the merging point of a path pair, we need to collect and construct constraints for a symbolized candidate distinguisher. To make sure that the unsatisfiability is resulted from the differing values of the candidate distinguisher, instead of from the path infeasibility, we first symbolically execute the two paths separately to make sure they both are feasible. Path pairs with infeasible paths will be directly discarded.

Given a candidate distinguisher, `V` of an extended path pair, we symbolize it into two different symbols for the two paths, e.g., $Symbol_{V_1}$ and $Symbol_{V_2}$. On the two paths, we then collect constraints related to `V` *separately*. After that, we further construct an artificial constraint, $Symbol_{V_1} == Symbol_{V_2}$. All these constraints are put together and then sent to a solver. We will receive a result from the solver, either `SAT` or `UNSAT`. If the result is `UNSAT` (unsolvable), we can tell that V_1 and V_2 must differ, given the conservativeness of under-constraint symbolic execution. In this case, we deem `V` as a distinguisher. This way, NDI precisely identifies distinguishers for the inconsistent path pair.

An example. To demonstrate the process of identification, we here to prove that `fname_p` in Figure 3 is a distinguisher over the extended path pair: the pre-path is ⑭-⑮-⑯-⑪-⑫, and the inconsistent path pair is path-1 ⑬-⑭-⑲ and path-2 ⑬-⑲. First of

all, `fname_p` will be symbolized to $Symbol_{v_1}$ on path-1 and $Symbol_{v_2}$ on path-2. On path-1, we collect the constraint $Symbol_{v_1}$!= $NULL$ because of the check on line 13. On path-2, we collect the constraint $Symbol_{v_2}$ == $NULL$ also because of the check on line 13. Finally, we generate an artificial constraint, $Symbol_{v_1}$ == $Symbol_{v_2}$, and send all those constraints to solver. It is obvious that the result of the solver is `UNSAT`, i.e., $Symbol_{v_1}$ can not equal to $Symbol_{v_2}$. Therefore, we conclude that `fname_p` is a distinguisher of the path pair.

4.3 Two-Staged Analysis for Inconsistency Recovery

The goal of NDI is to detect non-distinguishable or unrecovered inconsistencies as potential bugs, non-distinguishable inconsistencies indicate that the recovery of inconsistency is impossible, or the recovery does not happen before a use of the critical variable. Therefore, if an inconsistent path pair does not have any distinguishers, we can directly conclude that it is a bug. Otherwise, we will need to analyze possible recoveries of the inconsistency after the two paths merge. The analysis is to check (1) whether the distinguishers are overwritten, so the inconsistency clearly becomes non-distinguishable, or (2) whether the recovery is not performed at all before the use of the critical variable.

We find the recovery analysis challenging because it requires an inter-procedural, path-based, and field-sensitive analysis, which often suffers from path explosion or imprecision. To address this challenge, we perform a two-staged analysis. We observe that, given a path pair, typically there is only one critical variable, but there are multiple distinguishers. More importantly, the use of the critical variable is a requirement to form a bug. If the critical variable is not used at all, we do not need to analyze the distinguishers or recovery. Therefore, we propose to first leverage the uses of the critical variable to quickly filter out irrelevant code. In particular, in the first stage, we perform an efficient coarse-grained function-level program slicing against critical variables (not distinguishers) to limit our analysis scope; only functions that are along the paths from the merging point to the use of the critical variable are within the slice; all other functions are filtered out, which ensures scalability. In the second stage, we perform a fine-grained analysis against both critical variables and distinguishers (not targeted in the first stage), to precisely identify if distinguishers are used for recovery.

Coarse-grained function-level program slicing. In the first stage, we perform a coarse-grained function-level program slicing over critical variables. To improve the efficiency, the coarse-grained analysis is inter-procedural, flow-insensitive, and field-insensitive. The intuition of the first stage is that checks of distinguishers can only recover inconsistency before the uses of the critical variable. Therefore, if we can identify function paths that start from the function with inconsistency and end at the function with the uses of the critical variable, we can just focus the precise analysis of the second stage on the function paths, which can effectively reduce the analysis scope.

To locate the uses of a critical variable roughly but quickly, we need to address a few problems. For aliasing, we adopt the conservative `MayAlias` results. Note that the analysis is only applied to the critical variables, but not distinguishers. To balance the efficiency and accuracy, the coarse-grained analysis adopts a k-limiting analysis, which is commonly used by static analysis [18, 19, 29, 41]. Moreover, we use type analysis [24, 31] to find indirect-call targets.

Fine-grained recovery analysis. In the second stage, NDI performs fine-grained analysis within the program slice to determine whether distinguishers are used to recover inconsistencies. This analysis is an inter-procedural, path-based, flow- and field-sensitive one. Unlike the first stage, here we apply the precise analysis against both critical variables and distinguishers. We perform our precise analysis on each critical variable again because the analysis of critical variable in the first stage is imprecise. After that, we perform precise analysis against distinguishers to collect all possible recoveries. Recoveries are those checks (e.g. `if` statement) involving distinguishers or their aliases. NDI will report a bug if no recovery is performed before the uses.

An example. Now we use an example from `FreeBSD`, shown in Figure 4, to explain the two-staged analysis. This is a NULL pointer dereference identified by NDI. The return value of the function `ocs_hw_qtop_parse` in Figure 4 will be NULL on allocation failure of `qtop` on line 8. On the other path, `ocs_hw_qtop_parse` will return a pointer `qtop` which could never be NULL because of the check on line 6. In function `ocs_hw_setup`, the return value of `ocs_hw_qtop_parse` is assigned to `hw->qtop`, and there is a dereference of it after that without any check, which leads to a NULL pointer dereference bug.

```
1  ocs_hw_qtop_t *ocs_hw_qtop_parse(ocs_hw_t *hw, ...)
2  {
3      ocs_hw_qtop_t *qtop;
4      ...
5      qtop = ocs_malloc(...);
6      if (qtop == NULL) {
7          ...
8          return NULL;
9      }
10     qtop->os = hw->os;
11     ...
12     qtop->alloc_count = OCS_HW_MAX_QTOP_ENTRIES;
13     qtop->inuse_count = 0;
14     return qtop;
15 }
16
17 ocs_hw_rtn_e ocs_hw_setup(ocs_hw_t *hw, ...)
18 {
19     hw->qtop = ocs_hw_qtop_parse(hw, hw->config.queue_topology);
20     hw->config.n_eq = hw->qtop->entry_counts[QTOP_EQ];
21     ...
22 }
```

Figure 4: An example for explaining two-staged analysis.

In the first stage, we apply the coarse-grained analysis against the return value of function `ocs_hw_qtop_parse`, which identifies `hw->qtop` on line 19 in Figure 4 as its alias variable. As our analysis is field-insensitive, the analysis will further collect all field variables against `hw->qtop` like `hw->qtop->entry_counts`. Next, we identify the uses of the critical variable `hw->qtop`, which is a load operation on line 20. Finally, we determine the relevant call path is `ocs_hw_qtop_parse`—`ocs_hw_setup`, and our program slice only contains these two functions.

In the second stage, we perform a precise inter-procedural analysis against the return value of `ocs_hw_qtop_parse` and all its distinguishers. In our example, we will collect distinguishers on the inconsistent path pair: path-1 ⑥-⑧ and path-2 ⑥-⑩-⑭. Theses distinguishers include `qtop->os`, `qtop->alloc_count`, and the

return value itself since they hold differing values on different paths. After building all alias set against each distinguisher, we further identify possible checks which could recover the inconsistency. In this example, there is no such a recovery, so NDI identifies it as a non-distinguishable inconsistency and reports it as a bug.

Reporting and confirming bugs. Our bug report takes two formats. If there is no distinguisher, NDI will simply report the function name, the inconsistent path pairs along with the critical variable. If there are distinguishers, NDI will first report the function with inconsistency, the function with the use of the critical variable and the path between them on the call graph. In addition, it will report the critical variable and the use of the critical variables as supplementary information. We then manually check the path between the merging point of the path pair and the use of the critical variable to make sure there is no recovery and confirm the path is feasible. After confirmation, we report bugs to maintainers.

5 Implementation

We have implemented NDI based on LLVM 15, including a pass that translates IR into graph expressions, a pass that collects paths on functions and inconsistent path pairs, a pass that performs a path-based value flow analysis, a pass that collects distinguishers for inconsistent path pairs, and finally a pass that detects bugs related with non-distinguishable inconsistencies. The rest of this section presents important implementation details of NDI.

5.1 Data Structures and Representations

A path is represented with an ordered list composed of pointers to basic blocks. NDI then represents an inconsistent path pair with a combination of two paths. Furthermore, NDI represents critical variables and distinguishers with an unordered set composed of pointers to variables. Also, NDI uses a call graph to represent the searching scope of the two-staged analysis. This call graph consists of functions collected in the coarse-grained stage, and we use the number of nodes in the call graph to represent the scale of the searching scope.

5.2 Implemented Security States/Operations

To demonstrate how NDI works, we focus on three inconsistencies: memory release, NULL state, and initialization for the following reasons. (1) These inconsistencies exist widely in large systems and are often misused by programmers. (2) These inconsistencies can lead to severe security impacts such as memory corruption and denial-of-service. (3) Traditional analysis techniques such as fuzzing cannot easily find such bugs, which may locate in deep code paths. However, note that the types of inconsistency are not limited; NDI can be further extended to support more types of inconsistencies, such as boundary check, refcount, and lock/unlock.

Release inconsistency. Release inconsistency refers to that a variable is released on one path but not on the other. The release operation can be identified through deallocator functions, therefore to identify release inconsistency, the very first task is to collect deallocators. On each system, we manually identify the 10 most commonly used deallocators before collecting functions with release inconsistency. For collected functions, we take such variables whose values are released by deallocators as critical variables.

NULL state inconsistency. NULL state inconsistency refers to that in an inconsistent path pair, one path returns a non-NULL pointer while the other returns NULL. In practice, we will first collect functions with pointer-type return values. For each inconsistent path pair of these functions, we further confirm whether a return value equals NULL on one path and does not equal NULL on the other through under-constraint symbolic execution. The critical variable of NULL state inconsistency is exactly the return value itself. This NULL state inconsistency can also be easily extended to support other "invalid" pointers (e.g. a wild pointer composed of an error code).

Initialization inconsistency. Initialization inconsistency refers to that in an inconsistent path pair, a variable is initialized on one path but not on the other. In practice, for each inconsistent path pair, we first collect variables which are assigned on one path but remain unchanged on the other. Moreover, NDI supports other methods to identify initialization inconsistency. For instance, one can use function-based initializers like `memset` on one path but not on the other to indicate an initialization-inconsistent state.

5.3 Optimizations for Distinguisher Identification

§4.2 shows that NDI uses under-constraint symbolic execution to determine distinguishers. However, in the following situations, NDI can quickly decide the distinguishers, without the need for under-constraint symbolic execution, which improves performance.

Constant values. Sometimes the differing values of a distinguisher on two paths are constants, e.g. error code `ENOMEM` and NULL. In this case, NDI can simply compare these constants through arithmetic calculations without symbolic execution. For instance, once we identify that a variable is assigned with two different constant values on the pair, the variable can be regarded as a distinguisher.

Simple conflicting constraints. If constraints against a variable `v` are obviously unsatisfiable, e.g., one constraint is "`v == NULL`", and the other constraint is "`v != NULL`", we can immediately conclude that the constraints against `v` are unsatisfiable. Such cases are actually very common in the code. When identifying distinguishers, as explained in §4.2, we will build a constraint set against a variable. Before sending the set to the solver, we will quickly check whether there exist two constraints whose operands could be the same, but the predicate is inverse. Once we identify them, the variable is regarded as a distinguisher without symbolic execution.

5.4 Two-staged Analysis

Uses of critical variables. The first problem in two-staged analysis is to define what operations can be identified as uses of critical variables. It is obvious that `STORE` or `LOAD` operations on alias variables of the critical variable should be identified as uses, but we argue that the `GEP` operation should not be identified as uses, since this operation is just an address computation and will not cause a real security impact. However, this `GEP` operation may generate a wild pointer to an illegal address; therefore, `STORE` or `LOAD` operations on this wild pointer may lead to real bugs and should be regarded as uses. For instance, in our motivating example, there exists a wild pointer `&dh->params` in the function `DH_get0_pqg` of

Figure 1. The wild pointer results from a `GEP` operation on an alias variable `dh` and will not cause any real security impact. But once we detect a `LOAD` operation on the wild pointer `&dh->params` on line 42 in the function `ossl_ffc_params_get0_pqg`, we identify it as a use of the critical variable and further detect a bug.

Besides this, in some cases, function calls are also regarded as uses for critical variables. Though the passing critical variables itself will not lead to any security impact, we notice that some function calls, such as calling `memcpy`, are bound to cause severe problems as long as an alias variable is passed in without recovery. Therefore, we maintain a function list to record these functions. If an alias variable is passed into a function on the list, we will regard it as a use of the critical variables.

K-limiting analysis. In §4.3, we mentioned that NDI adopts a k-limiting method to form the entire scope of the coarse-grained stage analysis. Here we describe two implementation details. First, we need to decide a proper value for `k`. NDI currently takes `k` = 2 to balance accuracy and efficiency. In practice, if we take `k` = 1, NDI will miss many real bugs as many possible uses of critical variables are emitted. When `k` > 2, the two-staged analysis is less precise and will generate many false positive bug reports. Note that all the previous works [18, 19, 29, 41] will limit `k` to at most 2, which supports our choice for `k`. Second, we need to handle indirect calls. NDI adopts type analysis [24, 31] to match indirect calls with corresponding functions. However, we find that if the types are too general, the matching process will become very imprecise. To alleviate this problem, NDI will only resolve an indirect call when it has at least three arguments because such indirect calls can typically match only a few targets, and the results are very precise.

6 Evaluation

We evaluate NDI against well-tested and popular real-world programs, including `OpenSSL` library 3.0 (commit 21095479c0)[2], `FreeBSD` kernel 12[3], `httpd` server (commit 9dbdc1e517)[4] and `PHP` interpreter (commit ea62b8089a)[5]. The experiments are performed on a server with 256GB RAM and 32-core Intel E5-2670 CPU.

6.1 Analysis Performance

Table 1: Performance of NDI.

Targets	OpenSSL	FreeBSD	httpd	PHP
Num of Modules	886	1,483	189	376
Lines of Code	500k	16m	300k	1m
Analyzing Time	1h	2h	30m	1h30m
Peak Memory Usage	12.9G	28.1G	2.1G	27.3G

The evaluation results show that NDI can effectively analyze different-sized programs. Specifically, Table 1 shows the analyzing time, the project size and the peak memory usage of each program. Overall, we identify more than 4K functions with inconsistencies, 80K inconsistent path pairs and collect more than 10K distinguishers in total; the average searching scope of NDI is less than 100 functions with the help of scoping in the two-staged analysis.

[2]https://github.com/openssl/openssl
[3]https://github.com/freebsd/freebsd-src
[4]https://github.com/apache/httpd
[5]https://github.com/php/php-src

6.2 Bug Findings

NDI generates 86 bug reports, and it takes 70 person-hours in total to manually check and report them. Most time was spent on a small number of challenging cases that involve complicated code. For the majority of cases, the manual confirmation is quick because NDI provides a deterministic oracle, and the analysis scope is typically small. As shown in Table 2, we finally confirm 35, 7, 6 and 3 valid bugs from `OpenSSL` library, `FreeBSD` kernel, `httpd` server, and `PHP` interpreter, respectively.

Table 2: An aggregated list of bugs detected by NDI.

	OpenSSL	FreeBSD	httpd	PHP
Release inconsistency	18	3	0	0
Null state Inconsistency	14	4	1	3
Initialization inconsistency	3	0	5	0
Total	35	7	6	3

Most of these bugs can lead to security issues. Specifically, there are 8 use-before-initialization memory errors, 2 wild pointer dereferences (they can turn to arbitrary memory access if attackers control specific memory content on embedding systems without MMU), 23 null pointer dereferences, 14 memory leaks and 4 undefined behaviors. We have reported all of them to project maintainers. Until the submission of the paper, 46 bugs have been confirmed by maintainers, and 25 of them are already fixed in the latest version. Bugs are confirmed by maintainers through interactions on pull requests, issues, bugzilla and mailing. The detailed list of bugs is available in Appendix [27]. Note that many bugs in the list share the same bug ID since we report similar bugs as a batch in one report.

Bugs detected with multiple inconsistencies. Interestingly, several bugs can be detected with more than one inconsistency. There is 1 bug that can be detected by both release inconsistency and NULL state inconsistency, while there are 4 bugs that can be detected with release inconsistency and initialization inconsistency. Those bugs are categorized in the Table 2 by relevance. Note that although many functions may have different inconsistencies, not all of them can lead to the same bug. That is because NDI will identify different inconsistent path pairs and collect different critical variables over different inconsistencies; therefore, it will generate different program slices and finally report different bugs.

Security impact. NDI identifies 5 types of bugs: memory leak, NULL pointer dereference, wild pointer dereference, use before initialization and undefined behavior. The first four types, i.e., 92.2% of detected bugs, can cause different security impacts. 23 bugs and 2 bugs would cause NULL pointer dereference and wild pointer dereference, respectively, and those bugs would lead to Denial-of-Service (DoS) when triggered. 14 bugs would cause memory leaks, and they could cause memory exhaustion and DoS if they are triggered repeatedly. 8 bugs would cause use-before-initialization (UBI) memory errors, and the security impact of those bugs depends on what values the variables would hold without initialization.

We then use a fixed bug as a case study of a common security impact of the found bugs. In `OpenSSL`, the function `X509_STORE_new` returns NULL on a failure. However, in the function `create_cert_store`, the return value of `X509_STORE_new` is

used without NULL check. Therefore, it will lead to a NULL-pointer-dereference bug. If the bug is triggered, `OpenSSL` will crash and deny all the requests. This bug was identified by NDI; we reported the bug, and it has subsequently been fixed in the latest version.

Table 3: Precision of distinguisher identification and effectiveness of scoping. *I. Dist* = Number of identified distinguishers, *A. Dist* = Number of all distinguishers, *W/O S.* = # of functions to analyze without scoping, *W/ S.* = # of functions to analyze with scoping.

Name	I. Dist	A. Dist	W/O S.	W/ S.
DSO_free	4	7	43	38
evp_pkey_get_legacy	5	7	52	35
EVP_PKEY_copy_parameters	5	7	1,006	298
apreq_param_make	1	1	35	35
ocs_hw_qtop_parse	2	3	2	2
mock_srv_ctx_new	1	1	12	12
add_any_filter_handle	3	7	3	0
phpdbg_get_color	1	2	10	2
zend_arena_alloc	2	2	19,466	0
iwn_cmd	3	3	60	0

6.3 Identification of Distinguishers

To evaluate the accuracy of the identification, we first manually analyze the collected distinguishers. The corresponding results are shown in Table 3, column `I. Dist` (identified distinguishers) and `A. Dist` (all distinguishers). From such results, we find that every distinguisher identified by NDI is correct. This benefits from our precise under-constraint symbolic comparison. However, NDI is not complete in identifying distinguishers; In particular, NDI completely identifies distinguishers in 4 out of 10 cases. For the remaining 6 cases, NDI is not complete by missing a few distinguishers. Overall, NDI can identify 27 of 40 distinguishers, i.e., 67.5%. This is because NDI only performs intra-procedural distinguisher identification, and some distinguishers are manipulated by callees, thus can only be identified by inter-procedural analysis.

Interestingly, those missed distinguishers do not lead to any false positives in the final bug detection. This is because although many distinguishers can distinguish inconsistent path pair, only a few of them are actually used to recover the inconsistency. The actually used ones are typically simple ones (e.g., return values) and do not involve nested calls, so are captured by NDI.

For instance, in the function `evp_pkey_get_legacy`, we have mentioned that `pk->lock` should be a distinguisher in §2.1. `pk->lock` is not considered a distinguisher in NDI since `pk->lock` may not be always differing on the two paths from the intra-procedural analysis perspective, but missing the distinguisher `pk->lock` does not cause any false positive, because the actual used distinguisher is the return value of `evp_pkey_get_legacy` instead of `pk->lock`.

In general, we find that most distinguishers that are used for recovery can be detected intra-procedurally, which means our identification analysis is accurate enough in practice.

6.4 Effectiveness of Scoping

To evaluate the effectiveness of scoping (i.e., the first coarse-grained stage in the two-staged analysis), we disable it to make the comparison. With the scoping disabled, we directly perform the recovery analysis against critical variables and distinguishers. The analysis starts from the merging point of an inconsistent path pair. Note that in this analysis, we still need to collect alias variables over distinguishers and critical variables, and those variables are collected through the inter-procedural, path-based, flow- and field-sensitive analysis in the fine-grained stage.

Comparison with and without scoping. The results without scoping are on column `W/O S.`, and the results with scoping are on column `W/ S.` in Table 3. The result shows that scoping plays an important role in limiting the analysis scope for the precise recovery analysis. For instance, the scope of the function `EVP_PKEY_copy_parameters` would be more than 1k without scope analysis. By limiting the scope to less than 300, we can perform more precise alias analysis and find a bug in the `ssl_set_cert_and_key` function. Overall, the scoping reduces the number of functions to analyze by 98.0%. Particularly, in 3 of 10 cases, we reduce the scope size to zero; therefore, the fine-grained analysis stage can be completely omitted, making our analysis process much more efficient. Since there is no use at all, they would not lead to bugs.

6.5 Comparison with Most Related Tools

6.5.1 Comparison with Cross-Checking Tools. In this section, we compare NDI with two state-of-the-art similarity-based bug detection tools that are most related to NDI and open-sourced: IPPO [17] and Crix [20]. IPPO is a closely related work that is based on the similarity of paths. It identifies bugs by checking inconsistent security operations on two similar code paths. Crix detects bugs by checking similar code slices. It models and cross-checks the semantics of conditional statements in the peer slices of critical variables. First, we focus on how many bugs found by NDI can also be detected by them. In this evaluation, We feed the same bitcode files of all tested programs to the tools and compare ours with the bugs found by them.

Table 4: Comparison with state-of-the-art tools, in terms of bugs detected on all programs.

	OpenSSL	httpd	PHP	FreeBSD
NDI	35	6	3	7
IPPO	1	0	0	1
Crix	0	0	0	1

Bug missed by cross-checking. As shown in Table 4, only 3 bugs, i.e. 5.9% bugs, can be detected by cross-checking tools. And to make more clear comparison, we analyze bugs found in `OpenSSL` to prove that many bugs are theoretically unrecognizable by cross-checking. We choose `OpenSSL` because it is moderate and well organized, making the result more convincing. On `OpenSSL`, we manually collect all the functions with inconsistency or their wrappers, and compute the times they are misused. Those data are shown in Table 5. From Table 5 we notice that the misused functions in 26 of 35 bugs (i.e., 74.2%) have a misuse rate of more than 50%; therefore, they can not even be detected in principle by cross-checking because the inherent assumption of cross-checking—buggy code pieces are minority—just does not hold for them. Even worse, for some misused functions, there may be no correct use at all. For instance, the

function `DSO_free` contains a release inconsistency, i.e., the argument of `DSO_free` will be released on the normal path but not on the error paths. However, all 14 uses are buggy because they do not check the distinguishers of `DSO_free` at all. Finally, the inconsistency becomes non-distinguishable and causes memory leak bugs. Those bugs have been confirmed by the maintainers.

Table 5: Total Uses and Misuses about the targeted function in `OpenSSL`. *N1* = Total Uses, *N2* = Misuses, *R* = N2 / N1 * 100%. Cross-checking can only detect bugs when misuses are minority.

Function Name	N1	N2	R
DSO_free	14	14	100.0%
evp_pkey_get0_DH_int	3	2	66.6%
evp_pkey_get_legacy	18	13	72.2%
evp_pkey_get0_RSA_int	4	2	50.0%
OCSP_basic_add1_status	5	1	20.0%
X509V3_add_value	22	4	18.1%
EVP_PKEY_copy_parameters	18	2	11.1%
BN_bn2hex	9	1	11.1%
X509_STORE_new	21	1	4.7%

Trade-offs between NDI and cross-checking. NDI is not intended to replace cross-checking tools but to complement them. In general, both should be applied to broadly detect bugs in a target program. However, there are a few factors that affect the performance of NDI and previous tools, and users may consider them to prioritize the tools.

First, *the scale of the program.* NDI tends to outperform previous cross-checking tools on smaller programs. As mentioned in §2.2, cross-checking tools require a substantial number of similar code pieces for a statistical analysis. Smaller projects may not have enough similar code pieces. For instance, in the smallest project we tested, the `httpd` server, NDI finds 6 bugs; however, IPPO and Crix find 0 bugs. Second, *the customization level of the program.* A program with extensive custom code (e.g., to support special functionalities) may prioritize NDI. Even large programs such as the Linux kernel contain unique code logic to support custom features; we may not find enough similar code pieces for such code, and cross-checking would fail. Third, *the development process.* A program that is collaboratively developed by many programmers should also apply NDI. In this scenario, the designer of an API and the users of the API may have different expectations. An improperly-designed API may be frequently misused, but could be missed by cross-checking because the misuses are not minority cases. For example, in Table 5, most misused functions (74.2%) have a misuse rate of more than 50%, thus have been missed by cross-checking.

To summarize, a user may prioritize NDI in detecting bugs when the target program is small, customized, or collaboratively developed, and apply NDI together with existing cross-checking tools for other programs.

6.5.2 Comparison with Other Tools Detecting the Same Types. In §6.5.1, we compare NDI with the most related tools in terms of detection approach. In this section, we additionally compare NDI with state-of-the-art tools that detect the same bug types but may adopt a different detection approach. In particular, we select FICS [1] and IncreLux [44]. The current implementation of NDI can detect three types of bugs: memory leak, illegal pointer dereference, use before initialization (UBI). FICS can detect all the three types. It uses a machine learning–based bug detection technique that learns (correct) code patterns from the codebase and detect violations as bugs. IncreLux is a most recent work that focuses on detecting UBI bugs. It tracks the code paths to check whether a variable is fully initialized upon uses.

Table 6: Comparison with state-of-the-art tools detecting the same types.

	NDI	FICS	IncreLux
Memory leak	14	0	0
Illegal pointer dereference	25	1	0
Use before initialization	8	4	3

Bug missed by compared tools. As shown in Table 6, only 8 bugs, i.e. 15.6% of bugs, found by NDI can be detected by tools detecting the same types. Particularly, although IncreLux is specialized for detecting UBI bugs, it can only detect 3 out of 8 UBI bugs found by NDI. Though FICS claims to detect bugs with one-to-one inconsistency, the inconsistency has to be small enough to indicate a bug, which is not a must for NDI. Furthermore, IncreLux detects UBI bugs through type-qualifier analysis; however, the specifications of IncreLux are summarized based on a specific system (Linux kernel). Therefore, it does not perform well on other systems. On the other hand, we also collected the 8 bugs found by the compared tools from their papers. We found that NDI can rediscover 3 of them. We will explain why NDI misses other bugs in §6.5.3. The evaluation results show that (1) NDI and existing detection tools are complementary to each other, and (2) NDI finds more bugs in the tested programs.

Comparison on scalability. An important feature of NDI is that it localizes the potential buggy code (intra-procedural path pairs with inconsistencies) and focuses the analysis to the limited scope. Such a feature helps NDI avoid analyzing lengthy control/data flows and thus improve the scalability. As shown in Table 1, NDI is able to finish the detection from 30 minutes to 2 hours. For instance, FreeBSD contains 16 million lines of code, and NDI finished the detection within 2 hours, which is considered scalable.

In comparison, FICS and IncreLux may suffer from scalability because FICS adopts a machine-learning approach, and IncreLux has to keep track of object life-cycles from the beginning (i.e., allocation sites). In particular, we found that IncreLux consumes too much RAM; it fails when analyzing `OpenSSL` because of huge RAM usage (more than 250GB). NDI, however, completes all analysis with at most 30GB memory. Further, their detection costs a much longer time. For instance, FICS spends almost one week analyzing the programs, and takes about four days to analyze `FreeBSD` kernel alone. IncreLux spends about two days analyzing all programs, and takes one day to analyze the `FreeBSD` kernel. NDI, however, only spends about five hours completing all analysis, and two hours for FreeBSD. Overall, NDI outperforms these tools with a localized analysis technique that requires a much shorter detection time and consumes much less memory.

6.5.3 False Negatives To analyze false negatives, we first try to collect bugs (on the tested programs) found by compared tools. We

found it is impractical to collect the bugs by re-running the tools on the programs, because either they fail to run on some programs or there are too many false positives (e.g., IPPO has thousands of reports). Therefore, we collect the bugs from their papers. In total, we were able to collect 18 bugs. We first manually look into the bugs and found that, in theory, 14 of them can be detected by NDI's approach. As there are limitations with our implementation of NDI, we then apply NDI to the programs to see how many of them can be rediscovered in practice. It turns out that NDI can rediscover 8 of them, i.e. 44.4% of bugs. In the following, we look into the causes of the false negatives and summarize them.

No recovery after checks. NDI currently treats checks as a recovery of inconsistencies. However, checks over distinguishers are only a necessary condition of the recovery operation. NDI will not report a bug even if no actual recovery operation is performed after the checks. For instance, in `PHP`, IPPO finds a bug where the variable `preload_scripts` in the function `accel_preload` is released on one path but not on the other. However, NDI misses it because the return value of `accel_preload`, which is a distinguisher, is checked before uses of `preload_scripts`. Five false negatives fall in this category. As this is an implementation issue, it can potentially be eliminated in the future.

No inconsistency. Some bugs may not contain an inconsistency; therefore, NDI can not detect them. For instance, in `OpenSSL`, FICS finds a bug in `cms_RecipientInfo_pwri_crypt` where the variable `ec->key` is not released within the whole function. Since there is no release inconsistency, NDI cannot detect it. Four cases fall in this category. This is a problem with the NDI approach, and thus cannot be eliminated.

No critical variable uses. NDI focuses more on the detection precision and will not report uncertain bugs. For instance, FICS identifies a bug in `OpenSSL` that the return value of `BN_mod_inverse` should be checked within `RSA_X931_derive_ex`; however, NDI misses this bug because distinguishers and the return value keep alive beyond the searching scope, while there is no use against the return value within the scope. Only one case falls in this category.

6.6 False Positives

The overall false-positive rate of NDI is 40.6%, which we believe is promising for a static analyzer, and is better than the related tools [1, 17, 20, 44]. Even after they tuned the threshold to reduce false positives, IPPO still has a false-positive rate of 63.5%, and Crix still has the rate of 65.4%. the false-positive rate of FICS is 88.0% even after filtering. IncreLux randomly samples 44 bugs, and the rate is 50.0%. The rates of compared tools are higher because they are probabilistic while NDI is deterministic; Therefore, unlike NDI, the compared tools cannot guarantee the accuracy of their findings. The main causes of NDI's false positives are summarized below. Most causes are about implementation limitations instead of the approach of NDI.

Limitations of under-constrained symbolic execution. NDI employs under-constrained symbolic execution to identify distinguishers and remove infeasible paths. However, under-constrained symbolic execution is unable to remove all infeasible paths because of under-constrainedness. For instance, in `PHP` project, the function `zim_SplDoublyLinkedList_add` does not have a recovery against the NULL state inconsistency caused by the callee function `spl_ptr_llist_offset`. However, after careful confirmation of the maintainer, we find the return value of `spl_ptr_llist_offset` can never be NULL within the first function, which is not correctly identified by NDI. Such cases account for 52% of false positives.

Imprecise alias analysis. NDI attempts to identify uses of critical variables and checks over distinguishers by inter-procedural alias analysis. However, due to the alias analysis still has false positives, NDI will collect incorrect uses of critical variables and report false bug reports. Such cases account for 24% of false positives.

Others. Besides the above causes, false positives can also be caused by reasons like some recovery are omitted because of optimization and the checks of distinguishers are missed. Such cases account for 24% of false positives.

6.7 Bug Coverage of NDI's Approach

We emphasize again that NDI is designed to complement existing detection tools by covering their blind spots, but not to replace them. NDI is expected to have out-of-the-scope cases. However, we still want to get a sense of the coverage of NDI's approach. To evaluate the coverage of NDI, we perform an empirical analysis. With a set of real bugs, we analyze (1) how many bugs would be missed by NDI's approach in principle and (2) how many are missed by NDI's current implementation. We manually collect all memory bugs (in total 105 bugs are identified), which are fixed between 26 March, 2022 to 26 April, 2022 and in `OpenSSL` library, `httpd` server, `PHP` interpreter, and `FreeBSD` kernel. Those bugs are listed in Appendix [27]. We further remove 1 out of 105 bugs because code of the bug has been removed from projects. Finally, we have 104 bugs. Among those bugs, we identify 56 of 104 bugs, i.e., 53.8% of bugs, can be covered by NDI's approach. Then, we actually apply NDI to them and successfully detect 19 bugs. That is, NDI's implementation has 37 false negatives, which can be addressed with implementation improvements in the future.

7 Discussion

Applicability of NDI. NDI covers blind spots of traditional specification-based or deviation-based detection. However, NDI is never a complete detection approach that is applicable to all cases. In particular, NDI works with two pre-conditions in principle. First, the target function must have multiple code paths. the bug path must have another path to pair with. Second, there must be an inconsistent path pair. If both paths fail to perform the security operation, NDI will miss the bug. As confirmed in §6.7, NDI's approach has a high coverage/recall of 53% in principle.

Data-flow analysis. Although NDI has identified 51 new bugs by focusing on three security operations, our bug detection in principle should find more bugs than practice with a higher precision rate. One of the main reasons is that the data-flow analysis used in different phases has accuracy issues, especially due to aliasing, indirect calls, and assembly. In addition, in the coarse-grained scope analysis, we set a limit for the depth of analysis for path-explosion cases, which may introduce false negatives. Addressing such issues is an independent research topic. Once we have a more accurate data-flow analysis, NDI will also improve its efficiency and accuracy.

Supporting more security operations. NDI is a generic detection approach that can be plugged in with various security operations. In this paper, we use three operations as examples. In fact, the targeted security operations are an *input* to NDI. Other components, such as distinguisher identification, of NDI remain the same. Other security operations include bound checks, lock/unlock and reference counting. To support a new security operation, we will need (1) automated identification of the security operation in the code and (2) automated identification of the corresponding critical variable. One challenge is that, if it involves custom functions, we may need a pre-defined list of them. In addition, a critical variable is the one targeted by the security operation. This is typically easy to identify, except for lock/unlock where the critical variable is not targeted directly, which will need additional inference.

Comparison with Newton. A state-of-the-art tool, Newton [32], shares a similar logic when detecting vulnerable gadgets for ROP attack: It collects indirect call gadgets which could be used by ROP attackers through dynamic taint analysis. NDI has two significant differences compared with Newton: first, Newton is a dynamic tool while NDI is a static one. Second Newton focuses on ROP attack and only collects indirect call gadgets; however, NDI is a general bug detection oracle and do not focus on a particular gadget.

8 Related Works

Specification-based bug detection. In order to detect security issues in a target system, the most intuitive and classical method is to specify patterns and detect bugs against them. Smatch [4] and Coccinelle [25] match and identify specified source-code patterns, which are widely used to detect simple bugs in the Linux kernel. Recently, more analysis tools have been developed to identify bugs based on code patterns. LRSan [34] specifies lacking-recheck bugs, while deadline specifies and detects double-fetch bugs in OS kernels. SADA [3] detects unsafe DMA accesses in device drivers through a pattern-based analysis. DCUAF [2] performs a summary-based lock-set analysis to detect concurrency use-after-free bugs. HERO [38] precisely detects error-handling issues with a disordered calling pattern. Newton [32] uses a taint analysis to prevent ROP attacks with given target constraints. In general, the reliability of such methods depends on the quality of specifications and parsing the code accordingly, both of which can be complicated. On the other hand, specification-based detection methods usually perform well only on their target bug types, leaving new kinds of bugs undetected.

Inconsistency-based bug detection. Inconsistency-based detection methods report the inconsistencies among similar code snippets as bugs, known as cross-checking. Engler et al. [7] uses cross-checking to infer errors in systems code, while APISan [43] detects API misuses through cross-checking symbolic traces. Crix [20] uses indirect-call targets to construct semantically similar peer slices and further detects missing-check bugs. FICS [1] executes similarity and inconsistency analysis through a two-step clustering. IPPO [17] defines object-based similar paths with four static rules. EECatch [26] infers the appropriate severity level for error handling by analyzing it in specific modules, and further checks the exaggerated one. In general, cross-checking requires a sufficient number of similar code snippets to ensure reliability. Also, as we showed in §6, deviations may not be bugs. Different from previous inconsistency-based bug detection tools, our method does not rely on similarity analysis and can complement existing detection by covering their blind spots.

RID [21] detects refcount-leak bugs where a refcount operation appears only on one path. A fundamental difference is that RID does not consider distinguishers or recoveries. The requirement of RID is more restrictive than IPPO. That is, the two paths must share exactly the same return value and argument values, which is rare in practice. In comparison, NDI employs a custom under-constrained symbolic comparison to automatically and precisely detect distinguishers and a two-staged analysis to check recoveries. By manually checking the 51 new bugs found by NDI, we confirm that none of them can be detected by RID in principle because the paths always have differing return values or argument values.

Code-similarity analysis. Code-similarity analysis [11, 15, 16, 33, 40, 42] is a base of cross-checking and many bug detection methods. More recently, IPPO [17] proposes object-based similar paths with four static rules. FICS [1] applies machine learning techniques to measure the functional similarity of code snippets. MVP [39] uses signatures of code pieces to judge whether a piece of buggy code is similar to the existing code. Crix [20] regards indirect-call targets of the same indirect call as peers instead of computing the similarity directly. Usually, a code-similarity analysis method could only work well under limited scenarios or for specific bug types, and measuring the semantic-similarity of different code snippets remains unsolved.

9 Conclusion

Decades ago, researchers detect violations of specifications as bugs. Specifications can be too diverse and complicated to provide, and understanding the code and enforcing the specifications are also hard. Recent research favors cross-checking which detects deviations among similar code as bugs and does not require complicated specifications or code understanding. However, cross-checking suffers from bottlenecks; finding similar code is hard and even impossible for custom code, and bugs may not be the deviating cases. Thus, its detection is limited and unreliable. Motivated by the limitations, we proposed Non-Distinguishable Inconsistencies (NDI) as a deterministic oracle for detecting security bugs. The intuition is if inconsistent security operations are not distinguishable from the external, there is no way to recover them, which results in bugs. We realized NDI and ensured its precision and scalability through two new techniques. We use a custom under-constrained symbolic comparison to comprehensively and precisely detect distinguishers, and use a two-staged analysis to limit analysis scope and mitigate path explosion, and to precisely analyze recoveries. By applying NDI to the `OpenSSL` library, the `FreeBSD` kernel, the `httpd` server, and the `PHP` interpreter, we found 51 new bugs, and most of them could not be detected by the state-of-the-art tools.

10 Acknowledgment

We thank the anonymous reviewers for their valuable suggestions and comments. This research was supported in part by the NSF awards CNS-1815621, CNS-1931208, and CNS-2045478. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Mansour Ahmadi, Reza Mirzazade farkhani, Ryan Williams, and Long Lu. 2021. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2025–2040. https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi

[2] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 255–268.

[3] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. 2021. Static Detection of Unsafe DMA Accesses in Device Drivers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1629–1645. https://www.usenix.org/conference/usenixsecurity21/presentation/bai

[4] Dan Carpenter. 2009. Smatch - the source matcher. http://smatch.sourceforge.

[5] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. 2021. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *Proceedings of the Network and Distributed System Security Symposium*.

[6] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. *ACM SIGOPS operating systems review* 37, 5 (2003), 237–252.

[7] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72.

[8] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 72–82.

[9] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2014. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques* 10, 3 (2014), 211–217.

[10] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically Detecting Error Handling Bugs Using Error Specifications.. In *USENIX Security Symposium*. 345–362.

[11] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. *BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network*. Association for Computing Machinery, New York, NY, USA, 702–715.

[12] Yuan Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: Automated inference of error specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 472–482.

[13] Theodore Kremenek and Dawson R. Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *SAS*.

[14] Ted Kremenek, Paul Twohey, Godmar Back, and Andrew Ng. 2006. From Uncertainty to Belief: Inferring the Specification Within. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, Seattle, WA. https://www.usenix.org/conference/osdi-06/uncertainty-belief-inferring-specification-within

[15] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 201–213. https://doi.org/10.1145/2991079.2991102

[16] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. *aDiff: Cross-Version Binary Code Similarity Detection with DNN*. Association for Computing Machinery, New York, NY, USA, 667–678. https://doi.org/10.1145/3238147.3238199

[17] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. 2021. Detecting Missed Security Operations Through Differential Checking of Object-Based Similar Paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, 1627–1644. https://doi.org/10.1145/3460120.3485373

[18] Shen Liu, Gang Tan, and Trent Jaeger. 2017. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2359–2371.

[19] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[20] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1769–1786.

[21] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. 2016. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 531–544.

[22] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA.

[23] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[24] Ben Niu and Gang Tan. 2014. Modular Control-Flow Integrity. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, UK.

[25] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. *Acm sigops operating systems review* 42, 4 (2008), 247–260.

[26] Aditya Pakki and Kangjie Lu. 2020. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 1203–1218. https://doi.org/10.1145/3372297.3417256

[27] Zhou Qingyang, Wu Qiushi, Liu Dinghao, Ji Shouling, and Lu Kangjie. 2022. *Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs*. https://github.com/umnsec/ndi/blob/main/Nondistinguishable_Inconsistencies_as_a_Deterministic_Oracle_for_Detecting_Security_Bugs.pdf

[28] Lingyun Situ, Linzhang Wang, Yang Liu, Bing Mao, and Xuandong Li. 2018. Vanguard: Detecting Missing Checks for Prognosing Potential Vulnerabilities. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. ACM, 5.

[29] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[30] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations.. In *USENIX Security Symposium*. 379–394.

[31] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*. 941–955.

[32] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. 2017. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1675–1689.

[33] Shuai Wang and Dinghao Wu. 2017. In-Memory Fuzzing for Binary Code Similarity Analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 319–330.

[34] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check it Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[35] Westley Weimer and George C Necula. 2005. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 461–476.

[36] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*.

[37] Qian Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. 2011. Iterative mining of resource-releasing specifications. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 233–242.

[38] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2041–2058. https://www.usenix.org/conference/usenixsecurity21/presentation/wu-qiushi

[39] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1165–1182.

[40] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Xiaodong Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017).

[41] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-after-Free Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery, 327–337.

[42] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 01 (Apr. 2020), 1145–1152. https://doi.org/10.1609/aaai.v34i01.5466

[43] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 363–378.

[44] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V Krishnamurthy, Trent Jaeger, et al. 2022. Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium (NDSS'22)*.