IMPROVING STATIC ANALYSIS FOR SOFTWARE SECURITY AT COMPILE-TIME AND RUNTIME

A Dissertation

by

GANG ZHAO

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jeff Huang
Committee Members,	Alfredo Garcia
	Daniel A. Jiménez
	Guofei Gu
Head of Department,	Scott Schaefer

May 2022

Major Subject: Computer Science

ABSTRACT

Software security is a crucial factor in software development and maintenance. Static analysis approaches can help secure software in different ways. First, it can help identify vulnerabilities ahead-of run. For example, we can search vulnerable code in wild that are similar to buggy code in existing CVE databases, in which the program properties computed by static analysis are desired. For more complicated bugs, such as concurrency bugs, static analyses can infer more complex program properties, including the relation of pointers (i.e., alias analysis) in the program and the partial order between statements (e.g., happens-before relation), thus detect potential vulnerabilities. Second, static analyses can compute program properties (e.g., data-flow, control-flow) that we can check at runtime to achieve specific security goals (e.g., no control-flow hijack).

In this dissertation, we present three approaches of computing static program properties, combined with other methods, that improves the state-of-the-art for securing real-world software at compile-time and runtime. First, for the core of searching vulnerable code, measuring code similarity, we present a new approach that combines static program properties, data-flow and control-flow, with deep learning method. This is to address two limitations of existing techniques: scalability and imprecision. With deep neural networks, the classification is efficient on modern GPUs, while data-flow/control-flow only needs to be computed once for each code. And by leveraging the information in the encoded data-flow/control-flow and the training datasets, the deep neural network model can learn a good metric for measuring similarity between codes.

Second, we present a new approach for detecting atomicity violations in Rust programs, which is a kind of semantic bugs and one of the main source of Rust concurrency issues. We use alias analysis to compute if two atomic operations could access the same variable, and happens-before relation to compute if two such atomic operations could be executed concurrently. With observed heuristics from existing research study, we then statically check if a set of atomic operations from different threads could potentially cause an atomicity violation. This approach is evaluated to be effective in a benchmark of real world Rust programs with known atomicity violations. We also

find a potential new atomicity violation in a Rust project from Github.

Both the above two approaches detect bugs ahead-of run. However, real-world programs are almost impossible to be bug-free. Therefore, we present a new approach that effectively defends target program against non-control data attacks by checking statically computed program properties at runtime. Specifically, we introduce a new concept, origin, to abstract a piece of program path and all memory objects owned by it. We then compute all intended cross-origin data flow at compile-time, and instrument the program to check any violations to it at runtime. With an origin-based heap allocator, this approach incurs very low runtime overhead but is still effective on a benchmark of real-world programs with known CVEs.

DEDICATION

To my big family.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Jeff Huang, and all my labmates, Bozhen Liu, Shiyou Huang, Peiming Liu, Brad Swain, Siwei Cui, Yifei Gao, Bowen Cai, Luochao Wang, and all other students ever worked in our lab. Thank you for both technical help and research comments on my work.

I would also like to thank all my family members. Without your support, I could have never went through my Ph.D. program.

At last, I would like to thank all my committee members and all anonymous reviewers that ever reviewed my work. Your precious reviews and suggestions all contribute to this dissertation.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Motivation

Security is now a key concern in software development and maintenance, including browsers such as Chromium[9] and Firefox[11], Linux kernel, blockchain applications, etc. This is partly the result of lessons learned from lots of severe vulnerabilities reported in the past few years, such as the infamous Heartbleed in OpenSSL disclosed in 2014, which affected all mainstream Linux distributions and famous websites such as Stack Overflow.

The root cause of Heartbleed is shown in Figure 1.1. It is essentially an out-of-bound read bug. The vulnerable function `tls1_process_heartbeat` is called through an external event, which can be controlled by an attacker. In the vulnerable function, a buffer overread will be triggered when the attacker provides a payload length (retrieved by `n2s`) that is larger than the actual size of the request message. As a result, more data will be copied from memory in function `ssl3_write_bytes` and sent to the attacker.

```
1  int tls1_process_heartbeat(SSL *s) {
2    unsigned char *p = &s->s3->rrec.data[0], *pl;
3    unsigned int payload;
4    hbtype = *p++;
5    n2s(p, payload);
6    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
7    bp = buffer;
8    // out-of-bound read may occur here
9    r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
10         3 + payload + padding);
11 }
```

Figure 1.1: Heartbleed.

To prevent/mitigate such attacks, static program analysis plays an important role. As shown in Figure1.2, it can help in both ahead-of run and runtime stage. First, static analysis is widely used

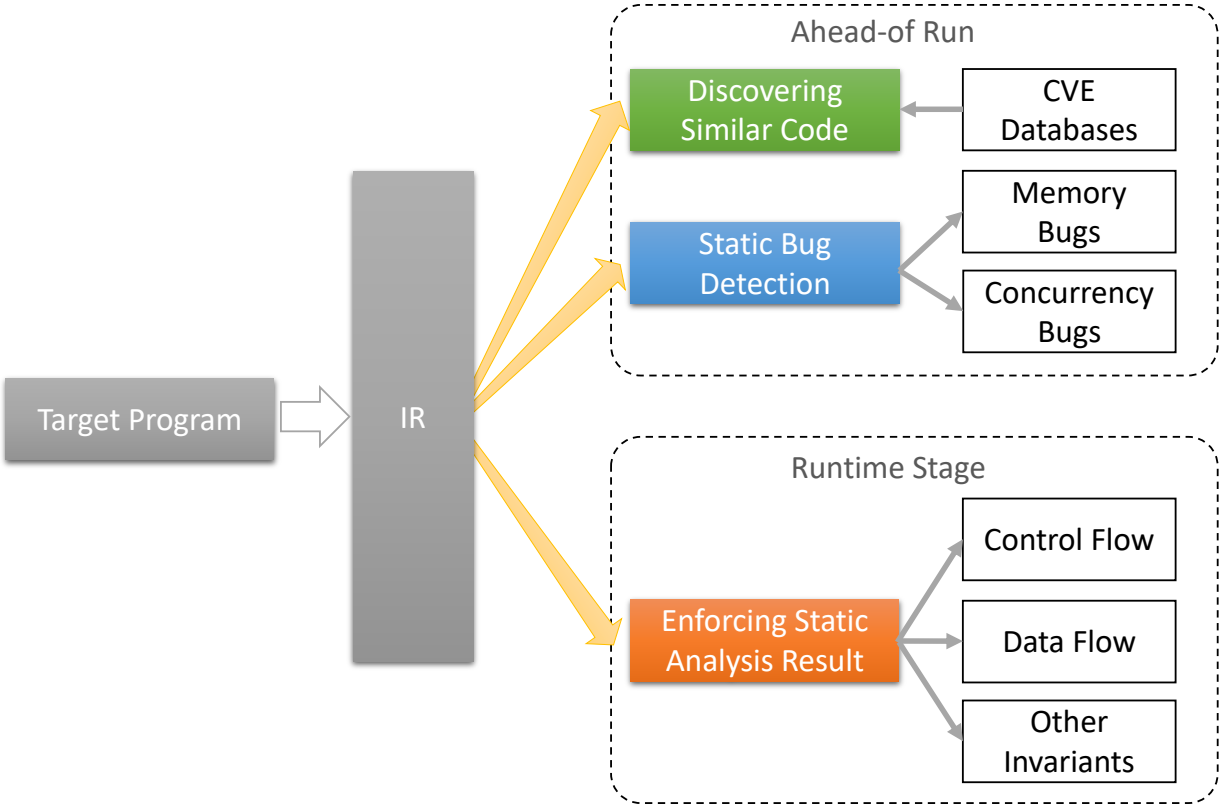


Figure 1.2: Static analysis for securing software. Note that approaches that require executing the program are usually treated as dynamic analysis. But there are some approaches relies on statically computed result and will instrument/rewrite the program (the effect could only be manifested at runtime). Such approaches are still considered as static analyses.

to detect bugs before running the program, mainly memory bugs and concurrency bugs [10, 43, 113, 42, 133, 82, 83]. With more and more bugs detected and CVEs reported every year, there are many CVE databases available (e.g., the national vulnerability database, the *CVE details* website, etc.). Based on these known CVEs, various approaches have been proposed to effectively search vulnerable code in wild programs [66, 47, 71, 46]. The core of such approaches is to determine whether the target code is similar to the known buggy code, for which the program properties computed by static analysis is crucial (e.g., data-flow, control-flow, program dependency graph, etc.).

Second, static analysis is also usually a perquisite of instrumentation/rewriting based approaches. Since it is practically impossible to achieve bug-free for any non-trivial program, in addition to

statically detecting bugs, many tools aim to prevent attacks that exploit unknown vulnerabilities at runtime. Such approaches often statically computes specific properties from the target program, and instrument the program to ensure that the properties are not broken, e.g., CFI (Control Flow Integrity) [13, 138, 140, 125, 28], DFI (Data Flow Integrity) [32, 118], CPI (Code Pointer Integrity) [75], etc.

This dissertation proposes three static analyses approaches, combined with other methods, that fit into components in Figure 1.2, significantly improving the state-of-the-art for software security. Figure 1.3 shows the high level design of the proposed approaches. First, we extract IR from the target program. For example, we use LLVM IR for C/C++ programs, MIR for Rust programs and WALA IR for Java programs. We then compute various static program properties from the IR, including data-flow/control-flow, alias relation and happens-before relation, which can be used to improve software security by combining other methods, such as deep learning, instrumentation, etc.

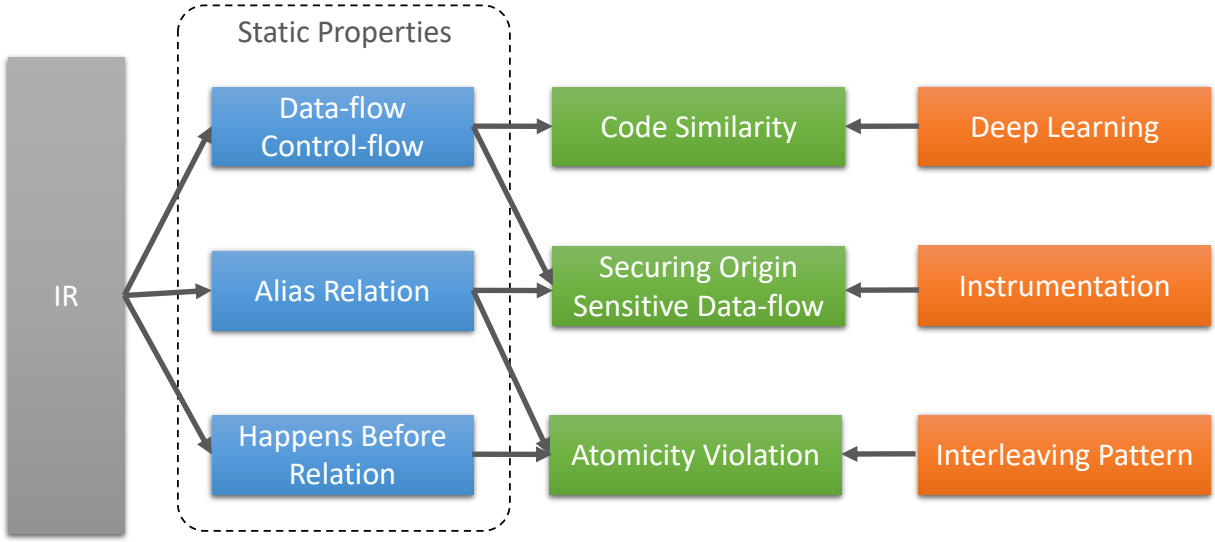


Figure 1.3: The high level design of approaches proposed in this dissertation. From the IR, we can compute various static program properties. Combining these properties with methods such as deep learning, instrumentation, we can detect atomicity violation, discover similar code and enforce origin sensitive data-flow at runtime.

More specifically, the first approach combines static data-flow and control-flow with deep learning method to measure code similarity, which is the core of discovering potentially vulnerable code with existing known CVEs. The second approach focuses on a specific category of bugs, atomicity violation, which is the main source of Rust concurrency issues. It computes statically a set of atomic operations from different threads that could access the same atomic variable at the same time, and applies heuristics (i.e., interleaving patterns) to determine whether they forms atomicity violation. The last approach enforces at runtime program properties inferred by static data-flow analysis through instrumenting the code, to prevent non-control attacks that exploit unknown vulnerabilities.

Next we first give an introduction to each of these approaches. Then we illustrate the concrete designs and experimental evaluations of them in the following chapters.

1.2 Code similarity

Measuring code similarity is the core of searching vulnerable code using existing CVEs. It is also fundamental for many other software engineering tasks, e.g., code clone detection [101], code reuse [70, 56], refactoring [53, 143], bug detection [80, 64]. With the large CVE database, it can be leveraged to identify vulnerable code used by other projects. While code syntactical similarity has been intensively studied in the software engineering community, few successes have been achieved on measuring code functional similarity. Most existing techniques [63, 131, 134, 80, 68, 105, 18, 19] follow the same pipeline: first extracting syntactical features from the code (in the form of raw texts, tokens, or AST), then applying certain distance metrics (e.g. Euclidean) to detect similar code. However, the syntactical representations used by these techniques significantly limit their capability in modeling code functional similarity. In addition, such simple distance metrics can be ineffective for certain feature spaces (e.g., categorical). Moreover, separately treating each code fragment makes it difficult to learn similarity between functionally similar code with different syntactics.

There exist a few techniques [74, 51, 84, 39] that construct a program dependence graph (PDG) for each code fragment and measure code similarity through finding isomorphic sub-graphs. Com-

pared to syntactical feature-based approaches, these graph-based techniques perform better in measuring code functional similarity. However, they either do not scale due to the complexity of graph isomorphism [35], or are imprecise due to the approximations made for improving scalability (e.g., mapping the subgraphs in PDG back to AST forest and comparing syntactical feature vectors extracted from AST [51]).

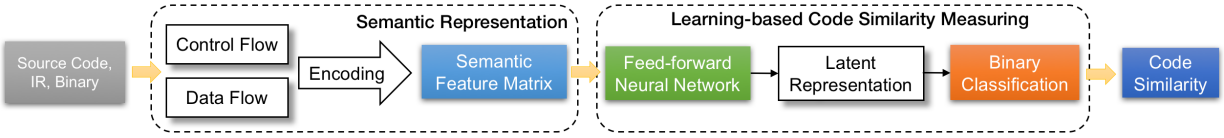


Figure 1.4: DeepSim System Architecture.

In this dissertation, we present a new approach, DeepSim, for measuring code functional similarity that significantly advances the state-of-the-art. DeepSim is based on two key insights. First, a feature representation is more powerful for measuring code semantics if it has a higher abstraction, because a higher abstraction requires capturing more semantic information of the code. Control flow and data flow represent a much higher abstraction than code syntactical features such as tokens [68, 105] and AST nodes [63]. Hence, DeepSim uses control flow and data flow as the basis of the similarity metrics. More importantly, we develop a novel encoding method that encodes both the code control flow and data flow into a compact semantic feature matrix, in which each element is a high dimensional sparse binary feature vector. With this encoding, we reduce the code similarity measuring problem to identifying similar patterns in matrices, which is more scalable than finding isomorphic sub-graphs.

Second, instead of manually extracting feature vectors from the semantic matrices and calculating the distance between different code, we leverage the power of deep neural networks (DNN) to learn a similarity metric based on the encoded semantic matrices. In the past decade DNN has led to breakthroughs in various areas such as computer vision, speech recognition and natural language processing [132, 117, 52]. Recently, a few efforts [114, 131, 20] have also been made

to tackle program analysis problems with DNN. We design a new DNN model that further learns high-level features from the semantic matrices and transforms the problem of measuring code functional similarity to binary classification. Through concatenating feature vectors from pairs of code fragments, this model can effectively learn patterns between functionally similar code with very different syntactics.

As depicted in Figure 1.4, DeepSim consists of two main components: (1) Code semantic representation through encoding control flow and data flow into a feature matrix; (2) Code similarity measurement through deep learning. The first component can take code fragments in any language as input in the form of source code, bytecode or binary code, as long as a data-flow graph and a control-flow graph can be constructed. The second component is a novel deep learning model containing two tightly integrated modules: a neural network module that extracts high-level features from the matrices of a code pair, and a binary classification module that determines if the code pair is functionally similar or not. These two modules work together to fully utilize the power of learning: the neural network module extracts latent representation for the classifier, and the signal propagated backward from the classifier, in turn, helps the neural network learn better representation from the input. Finally, the trained model can be used to measure code functional similarity.

1.3 Atomicity violation

Rust has been adopted in more and more industrial projects in the recent few years, e.g., Mozilla Servo browser engine, Solana blockchain, Amazon Firacracker for serverless computing, wasmtime for WebAssembly, etc. It provides comparable performance to C/C++, meanwhile guarantees memory-safety and data-race freedom [67]. Though these safety guarantees can be weakened by unsafe code (usually exists in libraries) [98], recent studies show that developers tend to use less unsafe code [15, 98].

When considering only pure safe code, however, concurrent bugs are still possible when using Rust. The reason is that Rust is only able to rule out data races but not high-level race conditions. According to [15], 15 out of the studied 38 non-blocking bugs appear in safe code, among which 5 are caused by incorrect usages of atomic operations, which are often used by develop-

ers to implement lock-free algorithms for better performance. In fact, atomic types provided in Rust standard/core library are now extensively used in popular open-source Rust projects (e.g., *parking_lot*, *tokio*, *Rocket*, etc.).

On one hand, since atomic operations (e.g., *add*, *sub* and *compare_exchange*) are translated to hardware atomic instructions, they are data race free. On the other hand, incorrect uses of them can still result in undesirable execution order across different threads, leading to race conditions. For example, assuming thread *t1* contains a load l^{t1} of atomic variable *v*, and thread *t2* contains a store s^{t2} of the same atomic variable. We may observe either the instruction sequence $\{l^{t1}, s^{t2}\}$ or $\{s^{t2}, l^{t1}\}$. If s^{t2} is intended to be run first to initialize resources, the first sequence may crash the program. In practice, a race condition usually invalidates a specific semantic property that programmers assume for the program, hence it is difficult to be precisely analyzed without the domain knowledge of the program. Similar to any other type of concurrent bugs (e.g., data races), testing is hard to find these atomicity violations due to the enormous interleaving space. Manually checking the source code could work but requires a huge amount of time and expertise of developers. Therefore, we propose to automatically detect such bugs in Rust programs, by leveraging Rust specific features and conventions.

To address the problem, an approximation is to consider a pair of atomic operations, and the code surrounded by the pair of atomic operations in the same thread, and also an atomic operation in another thread. Existing research [86] studied general atomicity violations in C/C++ by analyzing the serializability of different interleaving patterns. As shown in Table 1.1, four patterns are considered to be potential causes of atomicity violations. However, these patterns only consider the atomicity of the shared variable itself, but not that of the code regions. In this work, we will handle both.

Figures 1.5 and 1.6 show two examples simplified from real bugs. In function `update` shown in Figure 1.5, after loading the value of `NUM`, it runs some other code, and finally stores the updated value back into `NUM`. This works fine in a single thread scenario. However, in `main` function, another thread is created, which also updates `NUM`. Thus, the value of `NUM` can be either `1` or the

Table 1.1: Interleaving patterns of atomic operations that could potentially cause atomicity violations.

Access	Description	Problem
read *write* *read*	two reads interrupted by a concurrent write	The concurrent write makes two reads get different values for the same memory
write *write* *read*	write after read interrupted by a concurrent write	The local read may fail to get the expected values produced by the local write
write *read* *write*	two writes interrupted by a concurrent read	Invisible intermediate results are exposed to other threads
read *write* *write*	write after read interrupted by a concurrent write	The local read and write may have flow dependency, which could be broken by the write in the other thread

value computed at line 5. This falls into the last pattern in Table 1.1.

Apart from patterns listed in Table 1.1, a load operation performed by the second thread could also potentially cause atomicity violation, in which we need to consider the atomicity of the code region surrounded by the pair of atomic operations in the first thread. As shown in Figure 1.6, the function `init` is supposed to run only once or atomically to initialize resource. To achieve that, it first checks the value of `INIT`, if it is *false*, it calls `do_work`, then it updates the value of `INIT` to prevent the other thread from executing `do_work` again. Essentially the `load`/`store` of `INIT` here are intended to be used as lock/unlock, which is incorrect as they do not guarantee mutually exclusive accesses. When the other thread created in `main` function reaches line 4, the read value of `INIT` could be *false* if the main thread has not reached line 8. Therefore, `do_work` could be executed by two threads concurrently, which breaks the intended atomicity. To this end, we summarize a new interleaving pattern supplement to the four above, as shown in Table 1.2.

The overview of our approach, called MIRAV, is shown in Figure 1.7. Given a Rust project, we first get its mid-level intermediate representation, called MIR, including the MIR for the project itself and its dependencies. We choose MIR instead of LLVM IR since it has explicit type information of shared variable and no nested statements. Taking the MIR as input, we run our

```rust
static NUM: AtomicU32 = AtomicU32::new(0);

fn update() {
  let mut val = NUM.load(Ordering::Relaxed);
  val += get_increment(val);
  // Some other code that use val.
  do_work(val);
  NUM.store(val, Ordering::Relaxed);
}

fn main() {
  let t = thread::spawn(|| {
    NUM.store(1, Ordering::Relaxed);
  });
  update();
  t.join().unwrap();
}
```

Figure 1.5: Atomicity violation on the atomic variable.

Table 1.2: A new interleaving pattern of atomic operations that could potentially cause atomicity violations, which is supplement to Table 1.1.

Access	Description	Problem
read *read* *write*	write after read interrupted by a concurrent read	The local read and write may form an atomic code region that could be broken by the read (and its following code) in the other thread

inter-procedural alias analysis, which will provide the call graph and also the alias information of all atomic variables. Then we run a whole program analysis from the entry point of the project, in which we will collect all thread events and atomic operations and create a static happens-before graph. We intentionally ignore lock events (e.g., Mutex in Rust standard library) since we found that usually atomic operations are not used together with locks in the same scope after studying a few real world Rust projects (this aligns to the convention that programmers often use atomic types to implement lock-free algorithm to improve performance). We then choose a pair of atomic operations from one thread and another atomic operation from another thread, and check if it could

```rust
static INIT: AtomicBool = AtomicBool::new(false);

fn init() {
  if !INIT.load(Ordering::Relaxed) {
    // Code that should only be executed once
    // or executed atomically.
    do_work();
    INIT.store(true, Ordering::Relaxed);
  }
}

fn main() {
  let t = thread::spawn(init);
  init();
  t.join().unwrap();
}
```

Figure 1.6: Atomicity violation on the code region between a pair of load and store.

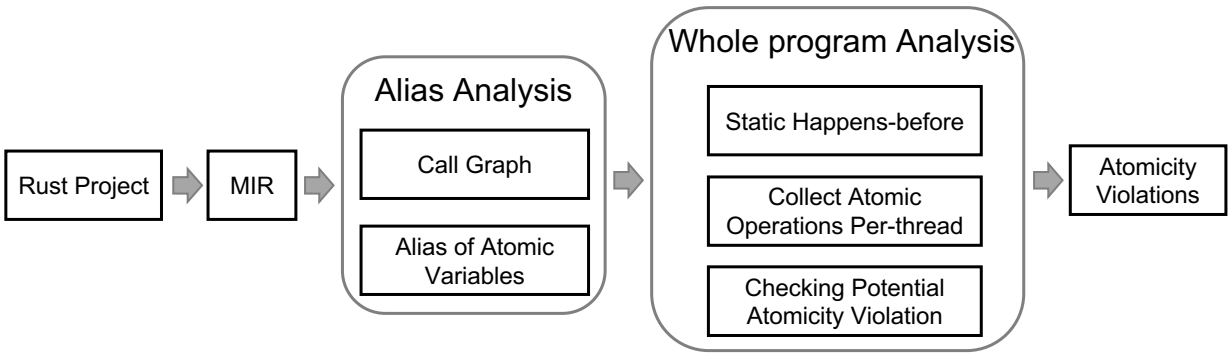


Figure 1.7: Overview.

be potentially an atomicity violation. If it is, we report its static call stack as well as the buggy code location.

1.4 Securing origin sensitive data-flow

Memory unsafe languages are susceptible to known attacks such as ROP [27], JOP [34, 26], COOP [110], etc. In the past decade, control-flow integrity (CFI) has gained much success with CFI solutions widely deployed (e.g., Google Chrome [5], Intel's CET [7], and more recently Mi-

crosoft's Control Flow Guard for Rust and LLVM compilers [8]). While this makes software more secure against control-oriented attacks, it may also make *data-only* attacks more prevalent.

Data-only attacks do not alter the program's control transfers but only change or leak the program's data stored in the heap. OpenSSL Heartbleed (CVE-2014-0160) [6] is an infamous example of data-only attacks, in which the vulnerable function `tls1_process_heartbeat` is called through an external event to process heartbeat messages controlled by an attacker. As a result, a buffer overread will be triggered when the attacker provides a payload length that is larger than the actual size of the request message, and more data will be copied from memory and sent back to the attacker. Similar vulnerabilities have also been found in other applications such as Google Chrome's PDFium module (CVE-2018-6120), by exploiting which, an attacker can corrupt PDF files in other tabs by opening a corrupted PDF file (under specific Chrome isolation policies).

None of the aforementioned data-only attacks can be prevented by CFI defenses, because these attacks target only *non-control* data. Surprisingly, existing data-flow solutions such as data-flow integrity (DFI) [32] and WIT [14] that leverage static analysis to restrict the set of objects that can be accessed by an instruction during runtime might also fail to prevent these attacks. It is because conservative static analysis techniques always fail to precisely compute the set of accessible objects and thus result in a larger accessible set than the program actually allowed. In the mean time, those techniques as well as other memory corruption detection tools such as AddressSanitizer [112] and memory safety techniques such as SoftBound [89] impose an extremely high overhead to the protected program.

We observed that to launch a data-only attack, there will normally be illegal data flows between logically isolated components of a program. For example, the illegal dataflow between PDFs opened in different tabs using PDFium as well as the illegal read of requests sent by other users in HeartBleed. The attack surface can be minimized by restricting different components of the program to only access a part of the entire address space. To achieve this, we present DFO, a new method that defends against such advanced data-only attacks by leveraging the concept of *origins*, which was previously proposed for static data race detection [83]. In the original work, it was

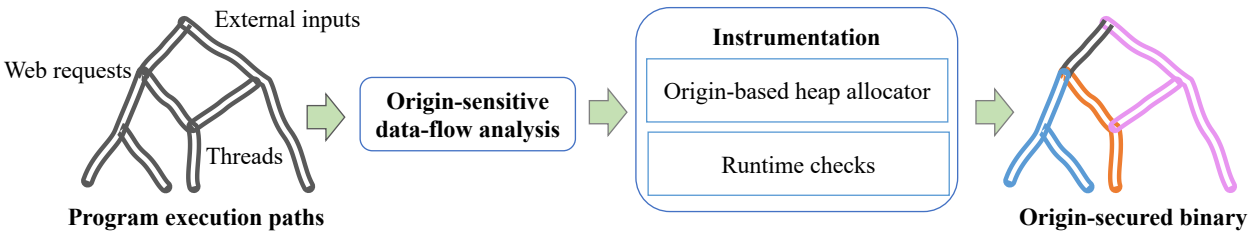


Figure 1.8: DFO overview. Paths in different colors represent secured different origins.

adopted to represent any entity that might lead to concurrency errors such as asynchronized event handlers as well as threads execution bodies.

In this dissertation, we further generalize the concept by allowing origins to represent arbitrary semantic items (with user-provided annotations). Cross-origin data-flow is first identified statically using origin-sensitive analysis, then any invalid cross-origin data-flow will be detected during runtime. The benefits brought by the use of origins are two fold:

- Unlike DFI, which enforces memory safety at instruction-level granularity, and ASAN/-SoftBound, which enforce memory safety at object-level granularity, DFO is more coarse-grained as it only detects memory corruption between different origins, which leads to a much lower runtime overhead.

- As already illustrated in [83], origin-sensitive analysis can precisely infer the cross-origin sharing information. Compared to DFI, which uses context-insensitive analysis, DFO can detect attacks that are missed by DFI due to the imprecision of static analysis.

Apart from applying origins for static analysis, we also introduce *dynamic* origins to precisely reason about the data-flow facts that are hard to compute by static analysis. A common example is event loops, in which the *same* handler is invoked repeatedly in the loop. Pure static analysis simply creates one origin to represent all the instances invoked in the loop and merges the data-flow between different iterations, where there are actually multiple origins being created *dynamically*. It is essential to distinguish these origins at runtime to ensure the integrity of cross-origin dataflows. For example, in HeartBleed, without dynamic origins, we cannot distinguish the data of different

iterations since they are all allocated at the same site, thus we would fail to prevent the overread.

The design overview of DFO is shown in Fig. 1.8. It has two components:

- At compile time, the relationship between data and origin (*i.e.*, which data belongs to which origin) is automatically determined by statically analyzing the execution flows originated from each origin. DFO uses origin-sensitive pointer analysis to find a complete set of heap objects owned by each origin, *i.e.*, the *bag* of the origin. Different origins may share heap objects, so bags can overlap. But each origin can only access a heap object in its own bag not any other bags (e.g., due to buffer overflow). Note that this type of analysis is different from taint analysis, because we do not taint data sources or propagate the taints; instead, we mark entry points such as thread start, event handler, or user input, and we perform pointer analysis to reason about origin-local and origin-shared heap objects. We call the analysis result an origin-data-sharing-graph (ODSG).

- At runtime, DFO allocates separate memory regions for different origins through an origin-based heap allocator, such that any cross-origin data flow can be detected efficiently. The design of DFO modifies only the dynamic memory allocator, without requiring any changes to the OS.

While there are some common APIs that can be identified as origins automatically (e.g., *pthread_create*), DFO requires minimal efforts from programmers to annotate appropriate origin entries for different applications. The application-specific knowledge from developers would help define an optimal set of origins for a stronger protection. Annotations (i.e., C/C++ custom attributes) can be easily inserted into the source code and minimal information is required from developers. Besides, any definition of origins is correct: it will never result in false alarms produced by DFO because the statically extracted cross-origin data-flow is an over-approximation.

2. RELATED WORK

2.1 Code Similarity

Code clone detection. According to [31, 102], code clones can be roughly classified into four types. *Type I-III* code clones only differ at tokens and statement level, while *Type IV* code clones are functionally similar code that usually have different implementations. Existing code clone detectors mainly target type I-III code clones through measuring code syntactical similarity based on representations such as text [18, 19, 100], tokens [80, 68, 105], abstract syntax tree [22] or parse-tree [63].

Su et al. [121] proposed to measure functional code similarity based on dynamic analysis. This approach captures runtime information inside callee functions through code instrumentation, but it may miss similar code due to the limited coverage of test inputs. Several other approaches have focused on scaling code clone techniques to large repositories [104, 105, 62, 123].

In addition, Gabel and Su [50] studied code uniqueness on a large code repository of 420M LOC, and observed significant code redundancy at granularity of 6-40 tokens. Barr et al. [21] studied patching programs by grafting existing code snippets in the same program and how difficult this grafting process is. Their result indicates a promising application of code similarity measuring techniques.

Machine learning for measuring code similarity. Carter et al. [31] proposed to measure code similarity through extracting handcrafted features from source code and applying self-organizing maps on them. Yang et al. [134] derived more robust features from source code based on inverse frequency-reverse document frequency, then applied K-Means to cluster code and predict the class for a new code fragment using the cosine distance metric. This approach cannot handle code fragments that do not belong to any existing clusters. Li et al. [79] also applied deep learning on code clone detection. They first identified tokens that are likely to be shared between programs, then computed the frequency of each token in each program. Given a code pair, they computed

a similarity score for each token based on their frequencies. The similarity score vector is then fed to the deep learning model to classify the code pair. Since token frequency is still a syntactical representation, the ability of this approach for measuring code functional similarity may be limited.

Semantic code search and equivalence checking. Reiss [99] used both static and dynamic specifications (e.g., keywords, class of method signatures, test cases, contracts) to search target code fragments. Stolee et al. [120] and Ke et al. [69] used symbolic execution to build constraints for each code fragment in the codebase and searched target code with input-output specifications. The returned code fragment should satisfy both the type signature and the conjoined constraints. This approach may not scale well as limited by symbolic execution and SMT solver. Moreover, Deissenboeck et al. [41] reported that 60%-70% of program chunks across five open-source Java projects refer to project specific data-types, which makes it difficult to directly compare inputs/outputs for equivalence checking across different projects. Partush and Yahav [96] presented a speculative code search algorithm that finds an interleaving of two programs with minimal abstract semantic differences, which abstracts the relationships between all variables in the two programs. This may not be applicable for non-homologous programs since there is not a clear mapping between most statements and variables.

2.2 Control/non-control Attacks and Defenses

Control attacks have a long history and many variants (e.g., ROP [27], JOP [34, 26], COOP [110]). These attacks alter the target program's control data (e.g., return addresses and function pointers) in order to execute injected malicious code or out-of-context library code (in particular, return-to-library attacks). A large number of control-flow integrity techniques [45, 129, 40, 138, 140] have been proposed to address the challenge of control-data attacks. For example, Tice et al. [125] implemented fine-grained, forward-edge CFI enforcement for GCC and LLVM with very low runtime overhead (1% to 8.7% on the SPEC CPU2006 benchmark suite). Their work only focuses on forward-edge control transfers, leaving many types of stack corruption to effective defenses that are already in use (e.g., ASLR, stack canary).

Code-pointer integrity (CPI) [75] prevents all control-flow hijack attacks through guaranteeing

the integrity of all code pointers (function pointers, saved return addresses) in the target program. It first detects sensitive pointers (code pointers and pointers that may be used to access sensitive pointers later) through conservative static analysis, then instruments the code to store all sensitive pointers in a safe region, with their metadata. Each dereference of a sensitive pointer is also instrumented to check its safety using the associated metadata. Since sensitive pointers are only a small sub-set of all pointers, CPI has very low runtime overhead.

Data-only attacks have been conceptually known for more than a decade. Chen et al. [36] constructed non-control-data exploits to show that data-oriented attack is a realistic threat. However, there exist no good solutions to defend them. More recently, FLOWSTITCH [60] proposes an approach to systematically generate data-oriented attacks that do not violate the control flow integrity. Data-oriented programming (DOP) [61] further develops Turing-complete non-control-data exploits for arbitrary x86 programs by reusing abundant data-oriented gadgets. Carlini et al. [29] proposed a non-control data attacks that leverage the memory corruption vulnerability to bypass fully-precise static CFI in five out of six real binaries.

Existing defenses for data-only attacks have focused on enforcing full memory safety or data-flow integrity. Cyclone [65] and CCured [91] extend C with a safe type system to enforce memory safety features. These approaches face the problem that there is a large (unported) legacy code base. SoftBound [89] with CETS [90] uses pointer bound checking to force a complete memory safety, at the cost of 2X-4X slowdown. Existing work enforces DFI by legitimate memory modification instruction analysis [32, 14] or through dynamic information tracking [136, 44]. However, DFI defenses are not yet practical, requiring large overheads, manual declassification or special hardware [119]. There are also some work target securing data through augmenting data space randomization or safe region through new hardware mechanisms [23, 88]. Software fault isolation (SFI) techniques [135, 87] can prevent attacks from escaping a sandbox and also allow the enforcement of high-level policies (e.g., restricting the allowed system calls of an application), however, they cannot prevent data-only attacks inside the application.

To reduce the overhead of data protection, researchers have also investigated the direction of

requiring assistance from users/programmers. YARRA [109] requires critical data types specified by users, then infers important data structures from those types to guarantee that such data can only be written by pointers with the given static types. DataShield [30] augments C/C++ with annotations that can be used to mark critical data types, and then prevents illegal reads and writes to those types at runtime. Samurai [97] proposes a memory model that requires users to identify the data to be protected, and uses replication and forward error correction to provide probabilistic guarantees of the critical memory.

A few work have also focused on data security of multi-threaded applications. Shreds [37] defines fine-grained segment of thread execution, each is associated with a protected memory pool. Invalid accesses from other threads or even the same thread are thus prevented. SMV [59] allows concurrent threads fully isolate their memory space in a controlled and parallel manner through enforcing least privilege memory views, with slight modifications to the OS kernel.

2.3 Rust concurrency bugs

Detecting concurrent bugs effectively and efficiently has long been a challenging problem that has been researched for decades. Various theories and tools have been proposed for the purpose. There are multiple different types of concurrent bugs including data races[111, 107, 93], deadlock [33, 57], starvation [108], atomicity violation [86, 85, 95], etc. Each of those bugs imposes unique challenges to detect them and have been tackled in different ways in the past.

Data race detection. As one of the most studied problems in the field, both static and dynamic tools have been developed to detect data races. For dynamic data race detection, lockset algorithm [142] and happens-before [48, 77] have already been thoroughly studied and widely adopted in commercial tools such as TSAN [111]. On the other hand, static data race detection remains to be a challenging problem [83, 25]. RacerD [25], developed by Facebook (now Meta) uses syntactic reasoning to reduce the false positives. O2 [83], on the other hand, performs an extensive whole program alias analysis to find potential data sharing between threads. Specially, Rust eliminates data races through its novelty type systems at compilation time [67].

Atomicity Violation Detection. Unlike data races/deadlock, which can be formally defined,

bugs like atomicity violation are considered to be semantic errors that can not be precisely detected without high-level semantic knowledge about the target program. Several static tools based on the theory of left/right mover [49, 106, 127] has been proposed. Most of those tools requires programmers' annotation to mark synchronization points in the programs, which makes those tools less desirable in practice. Other invariant-based approaches such as AVIO [86] detect atomictiy regions by observing access interleaving (AI) invariant automatically, which are learnt through "correct executions" and detects any potential violations in product runs.

Different from AVIO, MIRAI is a static tool. It also does not require user-provided annotations but instead depends on data dependencies to detect any potential interleavings that might cause the programs unserializable.

Static Tools for Rust. As the strong safety guarantees provided by the novel type system of Rust helps it gains popularity, more and more researchers have turned their attention to Rust in the past few years. Many tools have also been developed to help detect bugs in Rust programs. RUDRA [16], which targets at memory safety bugs in Rust, detect potential memory errors caused by the use of unsafe Rust with a set of predefined bug patterns. RUDRA has proven its practicality by discovering 264 previously unknown bugs. MIRChecker [81] uses static analysis and constraint solving techniques to capture the common pattern of Rust vulnerabilities.

Unlike RUDRA and MIRChecker, which aims to find bugs caused by the incorrect use of unsafe Rust, MIRAI targets at a more challenging problem. Since Rust can not provide safety guarantees for bugs like atomicity violation, MIRAI needs to analyze the target problem more extensively by examining both safe and unsafe Rust.

3. MEASURING CODE FUNCTIONAL SIMILARITY WITH DEEP LEARNING

[1] The main idea of this approach is leveraging the information provided by program data-flow and control-flow, encoding them into suitable data structure that can be efficiently fed to a deep neural network, and formalizing the similarity measuring problem as a binary classification problem. In this chapter, we will first briefly introduce the preliminaries of deep neural networks, then present the encoding and learning approach. At last, we will present the evaluation and the analysis of the results.

3.1 Preliminaries of Deep Learning

3.1.1 Feed-Forward Neural Network

Feed-forward neural network, also named multi-layer perceptron (MLP), is an universal yet powerful approximator [58], and has been widely used. It is essentially a mapping from inputs to outputs: $F : \mathbb{R}^{m_0} \rightarrow \mathbb{R}^{m_k}$, where m_0 is the dimensionality of inputs, m_k is the dimensionality of outputs generated by the last layer L_k. Each layer L_i is a mapping function: $F_{L_i} : \mathbb{R}^{m_{i-1}} \rightarrow \mathbb{R}^{m_i}, 1 \leq i \leq k$. F is hence the composed function of all its layers: $F = F_{L_k}(F_{L_{k-1}}(\cdots(F_{L_1}(x))))$.

The mapping function of each layer is composed by its units (called neurons). A neuron a_{ij} (the j-th neuron in layer i) takes the outputs generated by previous layer (or the initial inputs) as inputs, and calculates a scalar output value through an activation function g:

$a_{ij} = g(W_{ij}^T a_{i-1} + b_{ij})$ $\qquad\qquad$ $a_i = \{a_{i1}, ..., a_{im_i}\}$

where $W_{ij} \in \mathbb{R}^{m_{i-1}}, 1 \leq j \leq m_i$.

There are various activation functions according to different network designs. In this work we use ELU: $g(c) = c$ if $c \geq 0,\ e^c - 1$ if $c < 0$, where c is a scalar value [38].

The structure that several neurons form a layer and several layers form the integrated neural network has been proved effective in learning complicated non-linear mapping from inputs to

[1] Reprinted with permission from "DeepSim: Deep Learning Code Functional Similarity" by Zhao, Gang and Huang, Jeff, 2018. Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), Copyright 2018 by Gang Zhao.

outputs [54, 126]. Moreover, with the increasing computational power of modern GPUs, the depth of neural networks (the number of layers) can be made very large with a large training dataset – the so called "deep learning".

A representative feed-forward DNN model is deep autoencoder, which aims to learn a compact hidden representation from the inputs. Unlike standard encoding approaches (e.g., image/audio compression), it uses neural networks to learn the target encoding function instead of explicitly defining it. The goal is to minimize the error (e.g., squared error) of reconstructing the inputs from its hidden representation:

$$
\begin{aligned}
h_1 &= g(W_1 x + b_1) \\
&\cdots \\
h_k &= g(W_k h_{k-1} + b_k) \\
h'_{k-1} &= g(W' h_k + b'_{k-1}) \\
&\cdots \\
x' &= g(W'_1 h'_1 + b'_1) \\
J(x, x') &= \frac{1}{N} \sum_{i=1}^{N} ||x' - x||_2^2
\end{aligned}
\tag{3.1}
$$

where $W_i \in \mathbb{R}^{m_i \times m_{i-1}}$ and $b_i \in \mathbb{R}^{m_i}$ are the weights and biases of layer L_i ($W'_i = W_i^T$ if we use tied weights), h_k the encoded representation finally obtained, x the input of the model, x' the reconstruction of x, and N the number of all input samples.

Minimizing the reconstruction error forces this model to preserve as much information of the raw input as possible. However, some identity properties existed in the raw input may make the model simply learn a lookup function. Therefore, salt noise or corruption process is usually added to the input of the autoencoder, which is called Stacked Denoising Autoencoder (SdA) [126]. The corruption process works as randomly setting some components of the input to 0, which actually increases the sparsity of the autoencoder. The reconstruction process then tries to use this corrupted

input to reconstruct the raw input:

$$h' = g(W \cdot \varepsilon(x) + b)$$
$$x'' = g(W'h' + b')$$
$$J(x'', x) = \frac{1}{N}\sum_{i=1}^{N} ||x'' - x||_2^2 \tag{3.2}$$

where $\varepsilon(x)$ is the corruption process. In this way, it learns the hidden representation through capturing the statistical dependencies between components of the input.

3.1.2 Model Training

The training of neural networks is achieved by optimizing a defined loss function through gradient decent. For autoencoder, the reconstruction error discussed before is used as the loss function. For binary classification, a cross-entropy loss function is usually used:

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N} S^{(i)} log\tilde{S}^{(i)} + (1 - S^{(i)}) log(1 - \tilde{S}^{(i)}) \tag{3.3}$$

where $\theta = (W, b)$, $S^{(i)}$ is the ground truth label for sample i, and $\tilde{S}^{(i)}$ is the predicted label.

To optimize $J(\theta)$, we can apply gradient descent to update W and b:

$$\theta := \theta - \alpha \cdot \frac{1}{n}\sum^{n} \frac{\partial J(\theta)}{\partial \theta}$$

Here n is the size of batching samples for each update. The learning rate α can be used to control the pace of parameters update towards the direction of gradient descent. Decaying α with time is generally a good strategy for finding a good local minimum [72].

Given the large depth of DNN, it is ineffective to compute the derivative of the loss function separately for each parameter. Hence, *back propagation* [103] is often applied. It consists of three stages: 1) a feed-forward stage to generate outputs using the current weights; 2) a back-forward stage to compute the responsive error of each neuron using the ground truth outputs and feed-

forward outputs; and 3) a weights updating stage to compute the gradients using responsive errors and update the weights with a learning rate.

3.2 Encoding Semantic Features

We encode the code semantics represented by control flow and data flow into a single semantic matrix. In this section, I present the encoding process.

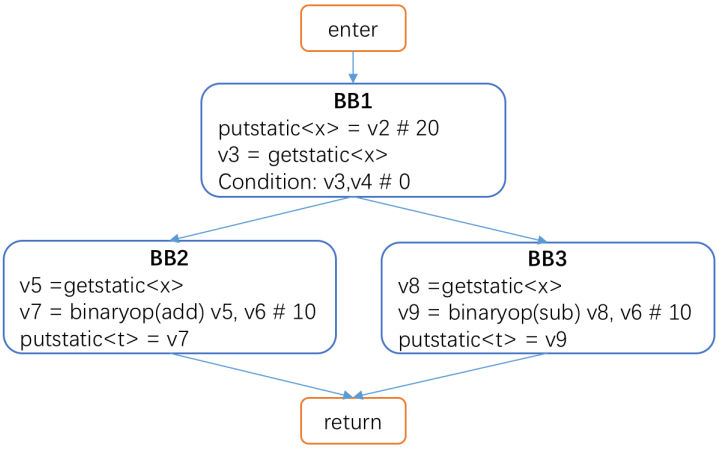
3.2.1 Control Flow and Data Flow

Control flow analysis and data flow analysis are two fundamental static program analysis techniques [92]. Control flow captures the dependence between code basic blocks and procedures, and data flow captures the flow of data values along program paths and operations. The information derived from these analyses is fundamental for representing the code behavior. The basic idea is to encode code control flow and data flow as relationships between variables and between basic blocks (e.g., operations, control jumps).

To obtain the control flow and data flow, we may perform the analysis on either source code, binary code, or any intermediate representation (e.g., Java bytecode, LLVM bitcode). In this thesis we focus on Java bytecode using the WALA framework [1].

```
1   public static int x = 0;
2   public static int t = 0;
3
4   public static void m1()
5   {
6       x = 20;
7       if (x!=0){
8           t = x + 10;
9       }else
10          t = x - 10;
11      }
12  }
```

(a) Example code



(b) Control flow graph of m1()

Figure 3.1: Control flow and data flow example.

Figure 3.1 illustrates a code example and its control flow graph (CFG). In the CFG, each rectangle represents a basic block, and in each basic block there may exist several intermediate instructions. For each basic block, a data flow graph (DFG) can be generated. Note that each variable and each basic block contain its *type* information. For example, x is a *static integer* variable, and $BB1$ is a basic block that contains a *conditional branch*. We describe how we encode them in more details in the next subsection.

3.2.2 Encoding Semantic Matrices

We consider three kinds of features for encoding the control flow and data flow information: variable features, basic block features, and relationship features between variables and basic blocks.

Variable features. For each variable, its type features contain its data type $V(t)$ (*bool, byte, char, double, float, int, long, short, void, other primitive types, JDK classes, user-defined types*), its modifiers $V(m)$ (*final, static, none*) and other information $V(o)$ (*basic, array, pointer, reference*). We encode them into a 19d binary feature vector $V = \{V(m), V(o), V(t)\}$. Consider the variable x in Figure 3.1a. It is a *static int* variable, so we have $V = \{0, 0, \mathbf{1}, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0, 0, \mathbf{1}, 0,$ $0, 0, 0, 0, 0\}$.

Basic block features. For each basic block, we consider the following seven types: *normal, loop, loop body, if, if body, switch, switch body*. Other types of basic blocks (e.g., *try catch* basic block) are regarded as $normal$ basic blocks. Similarly, we encode this type information into a 7d one-hot feature vector B. For example, in Figure 3.1b, BB1 is a *if* basic block and its representation $B = \{0, 0, 0, \mathbf{1}, 0, 0, 0\}$.

Relationship features between variables and basic blocks. We encode data flow and control flow as operations between variables and control jumps between basic blocks. In total, we identify 43 different operation types and use a 43d one-hot feature vector I to represent it[2]. To preserve the information of each operation between variables in the DFG, we derive a 81d feature vectors

[2]In our current implementation, we rely on WALA's instruction types to distinguish different operations and control jumps. In particular, for some instruction types (e.g., SSABinaryOpInstruction) that have more than one optional opcodes (e.g., *add, div, etc.*), we treat each opcode as a different operation.

$\mathcal{T} = \{V_{op1}, V_{op2}, I\}$, where V_{op1} and V_{op2} denote the variable feature vector for operand $op1$ and $op2$ respectively. To preserve the control flow, we extend $\mathcal{T}$ to be $\{V_{op1}, V_{op2}, I, B\}$ of size $E = 88$.

Now we formally define three rules to encode information for each relationship between variables and basic blocks:

- $\mathcal{T}(op1, op2) = \{V_{op1}, V_{op2}, I, B_{bb_{op1}}\}$ encodes the operations between two variables $op1$ and $op2$, as well as the type information of the two variables and the corresponding basic block. Here bb_{op1} denotes the basic block that $op1$ belongs to.
- $\mathcal{T}(op, bb) = \{V_{op}, V_0, I_0, B_{bb}\}$ encodes the relationship between a variable and its corresponding block. Here V_0, I_0 denote zero feature vectors.
- $\mathcal{T}(bb1, bb2) = \{V_0, V_0, I_0, B_{bb2}\}$ encodes the relationship between two neighbor basic blocks. Here $bb2$ is a successor of $bb1$ in the CFG.

Definition 1 (Semantic Feature Matrix)**.** *Given a code fragment, we can generate a matrix using the encoding rules above that captures its control flow and data flow information. Each row and column in $\mathbb{A}$ is a variable or a basic block. Their order corresponds to the code instruction and basic block order. $\mathbb{A}(i, j) = \mathcal{T}(i, j)$ is a binary feature vector of size $E = 88$ that represents the relationship between row i and column j, here $i, j = 1, 2, ..., n$, where $n = n_v + b_b$ is the summation of variables count and basic blocks count.*

Example. Consider the example in Figure 3.1 again. Its CFG has 11 variables (9 local variables and 2 static fields) and 6 basic blocks (including *enter, return, exit* block). From the encoding rules above, we can derive a sparse matrix as visualized in Figure 3.2.

Compared to the raw CFG and DFG, this semantic feature matrix representation has two advantages. First, it reduces the problem of finding isomorphic subgraphs to detecting similar patterns in matrices. Second, this structure (i.e., a single matrix) makes it easy to use for the later processes, such as clustering or neural networks in our approach.

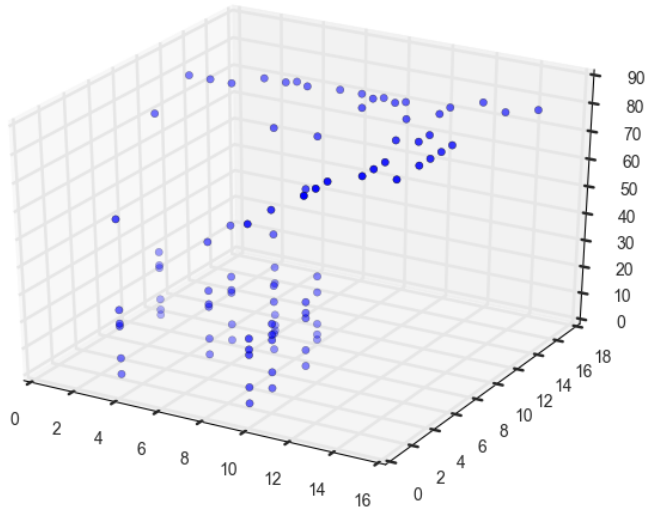


Figure 3.2: Semantic features matrix (17×17) generated from method *m1* in Figure 3.1. Along z axis are the 88d binary feature vectors. The value 1 is represented by a blue dot, and 0 represented by empty.

It is also worth noting that for most syntactically similar code that only differ in their identifier names, literal values or code layout, the generated semantic matrices will be identical, because these differences are normalized by CFG and DFG. This property ensures that our approach can also handle syntactical similarity.

We acknowledge that our matrix representation does not encode all information in a PDG. However, all dependence are implicitly encoded in our representation. For example, the code $y = x; z = x$ contains an input dependence; it is encoded as $\mathcal{T}(y, x)$ and $\mathcal{T}(z, x)$. Other types of dependence are encoded in similar way.

3.3 Learning-based Code Similarity Measuring

From the semantic matrix encoding described in the previous section, we can see that the semantic matrices generated from two *functionally* similar methods with syntactical differences may not be identical. We cannot directly use simple distance metric (e.g., Euclidean distance) for measuring their similarity, since we have no knowledge about whether some elements (i.e., feature vector $\mathcal{T}$) in a semantic matrix is more important than another. Moreover, different elements may be functionally similar (e.g., *binary add* operations between two *int* and *long* variables respec-

tively), but it is difficult to detect this similarity directly on the semantic feature matrix.

To address this issue, I design a deep learning model that can effectively identify similarity patterns between semantic matrices. In particular, I train a specially designed feed-forward neural network that generates high-level features from semantic matrices for each code pair, and classify them as functionally similar or not using the classification layer at the end of the neural network. This design eliminates the need to define a separate distance metric on the learned features. More importantly, this model can be finely trained in a supervised manner to learn better representation from the input through backward propagating the label signal of the classification layer (i.e.,{0, 1}, see Eq. 3.3). This helps the model effectively learn patterns from similar code with very different syntactics.

3.3.1 Features Learning

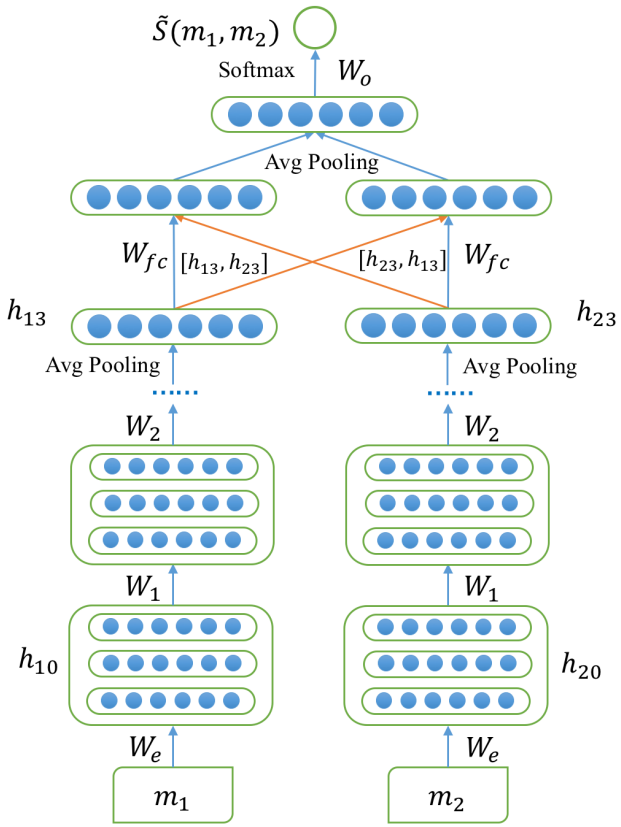


Figure 3.3: A schematic of our feed-forward model for measuring code functional similarity.

The architecture of our model is shown in Figure 3.3. As a static neural graph, this model requires a fixed size of input semantic feature matrices. Similar to recurrent neural network [55], we process all input matrices to a fixed size $k \times k$ ($k = 128$ in this work). For smaller methods, we pad their feature matrices with zero value. For larger methods, we truncate the matrices through keeping all CFG information and discarding part of DFG information that exceeds the matrix size (the original feature matrix size is $n = n_v + n_b$, after truncation, $k = n_{v'} + n_{b'} = 128$, where $n_{b'} = n_b$, $n_{v'} = 128 - n_b$).

Recall that we encode data flow and control flow into 88d sparse binary feature vectors $\mathcal{T}$. Similar to word embedding [24], we first map $\mathcal{T}$ into hidden state h_0 of size $c = 6$. This layer makes it possible to learn similarity between two relationship feature vectors. For example, *binary add* operations between two *int* and *float* variables respectively may share some similarities.

Handling code statement re-ordering. We then flatten each row of $\mathbb{A}$ to a vector with length of $c \cdot k$. We add two fully connected layers taking the flattened row feature vectors as inputs, in order to extract all information related to the variable or basic block represented by each row. At last, a pooling layer is added to summarize all the row feature vectors. To mitigate the effect of code statement re-ordering, we use average pooling instead of flattening the entire matrix.

```
1 int c = 2;
2 int x = 2*c;
3 int y = 3*c;
4 x = 2*y;
5 y = x*y;
```

```
1 int c = 2;
2 int y = 3*c;
3 int x = 2*c;
4 y = x*y;
5 x = 2*y;
```

Consider the two small programs above, which have identical code statements but different statement orders at lines 2 and 3 and between lines 4 and 5. The different statement orders at lines 4 and 5 lead to different x and y values at the end of the two programs. In our matrix representation, we encode all the data flow between the three variables, and the statement order decides which variable or basic block that each row of the matrix represents. Thus, we generate two different $\mathbb{A}$s. However, if we only consider the statements at lines 1-3, the two programs should be equivalent,

because there are no dependence between the two statements at lines 2 and 3.

Therefore, to make our model more robust in handling the statement re-ordering, we ignore the order of independent variables or basic blocks and keep only the order of variables or basic blocks with dependence between them. This can be achieved by performing average pooling on all those neighbor rows of $\mathbb{A}$ that represent independent variables or basic blocks, and flattening the rest rows (the order is preserved in the flattened vector). However, this would lead to various flattened vector lengths for different methods, which is difficult for a static neural network model to handle. Instead, we make approximation by directly performing averaging pooling on all rows, and leaving the rest task to our deep learning model.

Interestingly, another advantage of this design is that it reduces the model complexity, because the pooling layer reduces the input size from $c \times k \times k$ to $c \times k$ without any extra parameters (similar to pooling layers in convolutional neural network [78]). Thus, the training and predicting efficiency of our approach is increased.

Similar to the vanilla multi-layer autoencoder (recall Section 3.1.1), this model can be trained in an unsupervised manner through minimizing the reconstruction error (Eq. 3.1 and Eq. 3.2). The training consists of two phases. Each layer is first pre-trained to minimize the reconstruction error, and then the entire model is trained through minimizing its overall reconstruction error. In this way, the model can learn a non-linear mapping from the input $\mathbb{A}$ to a compressed latent representation.

However, the model trained by this approach may discriminate two similar but not identical inputs, e.g., two array search methods that respectively handle *int* and *float* arrays. To enforce the model to learn similar patterns across different samples, we utilize supervised training, which is achieved by the binary classification layer that takes as input the concatenation of the final latent representations of two methods. We describe it in more details in the next subsection.

3.3.2 Discovering Functional Similarity

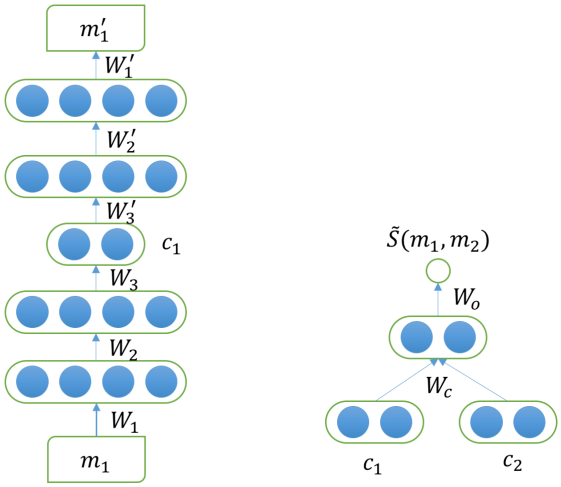
Given two methods with learned latent representations, a straightforward way to measure their functional similarity is calculating their distance in the hidden state space. This metric is applied by many existing techniques [63, 131].

However, such a simple distance metric is not satisfiable under two considerations. First, users have to perform heuristic analysis to find a suitable distance threshold for *every* dataset. Euclidean distance for measuring code similarity may not be applicable in the hidden space. For example, in the XOR problem, if we use Euclidean distance, the input (0, 0) would be closer to (0,1) and (1,0) than (1,1), which is a wrong classification. Second, a deterministic distance metric cannot leverage existing training dataset to finely train the model. Though it is feasible to obtain an optimal distance threshold using a training dataset, the parameters of the neural network model itself are not updated in this process.

I propose an alternative approach that formulates the problem of identifying functionally similar code as binary classification. More specifically, as shown in Figure 3.3, we concatenate the latent representation vectors h_{13} and h_{23} from methods m_1 and m_2 in different orders: $[h_{13}, h_{23}]$, $[h_{23}, h_{13}]$. Then we apply a fully connected layer on the two concatenated vectors with sharing weights W_{fc}. Another average pooling is performed on the output states, and finally the classification layers are added. We can now use cross-entropy as the cost function (Eq. 3.3), and minimize it to optimize the entire model. In this way, our model is able to learn similarity between each code pair with different syntactics.

Optimization. The two concatenations with different orders and the average pooling operation are important for optimizing the efficiency of our model. Note that as a similarity measuring task, we have a symmetric property: $S(m_i, m_j) = S(m_j, m_i)$. If we use only one concatenation $[h_{i3}, h_{j3}]$ or $[h_{j3}, h_{i3}]$, we have to add both (m_i, m_j) and (m_j, m_i) into our training dataset to satisfy the symmetric property, which will lead to nearly 2X training and predicting time.

When applying this model to discover functionally similar methods on a code repository of size N, we would need to run it $\frac{N(N-1)}{2}$ times. Considering that the classification layers only contain very few computations of the entire model (less than 1% matrix multiplications for our trained model), the efficiency can be significantly improved (almost 2X speedup) by separating the model into two parts during prediction. More specifically, for the N methods, we run the feed-forward phase of the model to generate their latent representation h_3. Then for each method pair, we only



(a) SdA for learning latent feature representation

(b) Measuring code similarity using latent representation

Figure 3.4: SdA baseline models. m_1 is a semantic matrix $\mathbb{A}$ generated from a method, c_1 and c_2 are two feature vectors generated by SdA from a method pair. For SdA-base, we put the two parts together when training, while for SdA-unsup, we do not back-propagate the error signal of (b) into (a).

run the classification phase.

Baseline Model. As a comparison, I also train an SdA as the baseline model, as depicted in Figure 3.4. In the baseline, we try to flatten each matrix to a long input vector of size $k \times k \times E$ (i.e., 1,441,792 if $k = 128$ and $E = 88$). However, even we set a small size of the first hidden layer (e.g., 100), there are over 1.44 $\times 10^8$ parameters, which makes the training difficult (unlike SdA, our model applies learning on each row followed by a row average pooling, hence its layer size is limited). Therefore, we slightly change the feature vector $\mathcal{T}$. As $\mathcal{T} = \{V_{op1}, V_{op2}, I, B\}$, $V_{op} = \{V(m), V(o), V(t)\}$, we use a 8-bit integer to represent each component of it, thus we drive a $\mathcal{T}'$ of size 8 (this can be understood as a handcrafted feature vector, as the characteristic vector in [63]). The other parts follow the standard SdA model.

I develop two settings for the SdA baseline model: *SdA-base* and *SdA-unsup*. SdA-base also adds classification layers on top of it (combine the two models in Figure 3.4), taking the concatenation of two latent representation vectors as inputs (with only one concatenating order). Thus,

the fine training step will also optimize the feed-forward encoding part of the model. SdA-unsup uses a separate classification model with one hidden layer, as shown in Figure 3.4b. It works by estimating an optimal similarity metric in the hidden space. In both cases we apply the same loss function in Eq. 3.3. The goal of these two settings is to present a comparison between our model and the vanilla SdA, and to understand the effectiveness of the binary classification formulation.

3.4 Experimental Evaluation

3.4.1 Datasets

We evaluated DeepSim on two datasets: Google Code Jam (GCJ) [3] and BigCloneBench [124]. In the first experiment, we follow recent literature [128, 121, 122] and use the programs from the Google Code Jam competition [3] as our dataset. We collected 1,669 projects from 12 competition problems, as shown in Table 3.1. Each project for the same problem is implemented by different programmers, and its correctness has been verified by Google. We have manually verified that the 12 problems are totally different. Therefore, programs for the same problem should be functionally similar (i.e., label 1), and those for different problems should be dissimilar (i.e., label 0). To better understand this benchmark, we manually inspected 15 projects for each problem and found that very few projects (6 pairs in total, 2 each in *Mushroom Monster* and *Rank and File*, and 1 each in *Brattleship* and *Senate Evacuation*) can be classified as syntactically similar[3]. Note that we did not remove any of these syntactically similar code, because our approach is also capable of handling them (as explained in Section 3.2). In addition, we found that more than 20% of the projects contain at least one obvious statement re-ordering. Most of them are caused by different orders of variable definitions used in loops or parameters for functions such as *min*, *max*, etc.

In the second experiment, we run DeepSim on the popular BigCloneBench dataset [124]. BigCloneBench is a large code clone benchmark that contains over 6,000,000 tagged clone pairs and 260,000 false clone pairs collected from 25,000 systems. Each clone pair in BigCloneBench is also a pair of methods, and is manually assigned a clone type. The total number of clone pairs obtained

[3]Our criterion for syntactically similar code is very relax: the number of different lines of code can be up to 15 and no structural difference.

Table 3.1: The Google Code Jam dataset.

Dataset statistics	Value
Projects	1,669
Total lines of code	98,117
Tokens	445,371
Vocabulary size	4,803
Similar method pairs	275,570
Dissimilar method pairs	1,116,376

from the downloaded database is slightly different from that reported in [124]. We discard code fragments without any tagged true or false clone pairs, and discard methods with LOC less than 5. In the end we obtained 50K methods, including 5.5M tagged clone pairs and 200K tagged false clone pairs. Most true/false clone pairs belong to clone type WT3/T4.

3.4.2 Implementation and Comparisons

For DeepSim and the SdA baseline models, we use WALA [1] to generate CFG/DFG from bytecode of each method, and construct the semantic feature matrix as described in Section 3.2. For efficiency, we store all the matrices in a sparse format. We use TensorFlow [2] to implement the neural networks models. For the GCJ dataset, we developed a plug-in in the Eclipse IDE to automatically inline source code file of each project to a single method. For BigCloneBench, because it does not provide the dependency libraries for the source code files, we modified WALA and the Polyglot-compiler framework to generate CFG/DFG for these source code files directly. For any external class type, we replace it with a unique type *Java.lang.Object*. This results in failures to get all fields in some cases. However, our tool is able to process over 95% of the clone pairs in BigCloneBench.

We also compared DeepSim with the three other state-of-the-art approaches: DECKARD [63], RtvNN [131] and CDLH [130]. DECKARD is a classical AST-based technique that generates characteristic vectors for each subtree using predefined rules and clusters them to detect code clones. In our experiment we used the stable version available in Github [4]. RtvNN uses recursive

Table 3.2: Parameter settings for different tools.

Tool	Parameters
DECKARD	Min tokens: 100, Stride: 2, Similarity threshold: 0.9
SdA-base	Layers size: (1024-512-256)-512-128, epoch: 6 Initial learning rate: 0.001, λ for L2 regularization: 0.000003 Corruption level: 0.25, Dropout: 0.75
SdA-separate	Feed-forward phase: Layers size: 1024-512-256, epoch: 1, λ_1 for L2 regularization: 0.0003, Corruption level: 0.25, initial learning rate: 0.001, Classification phase: Layers size: 512-128, epoch: 3, Dropout: 0.75, Initial learning rate: 0.001, λ_2 for L2 regularization: 0.001
RtvNN	RtNN phase: hidden layer size: 400, epoch: 25, λ_1 for L2 regularization: 0.005, Initial learning rate: 0.003, Clipping gradient range: (-5.0, 5.0), RvNN phase: hidden layer size: (400,400)-400, epoch: 5, Initial learning rate: 0.005 λ_1 for L2 regularization: 0.005 Distance threshold: 2.56
DeepSim	Layers size: 88-6, (128x6-256-64)-128-32, epoch: 4, Initial learning rate: 0.001, λ for L2 regularization: 0.00003 Dropout: 0.75

neural networks [117] to measure code similarity using Euclidean distance. Different from our approach, their model operates on source code tokens and AST and learns hidden representation for each code separately (i.e., unsupervised learning) rather than combining each code pair to learn similar patterns between them. Since the RtvNN implementation is not available, we implemented the approach following the paper [131]. CDLH is a recent machine learning-based functional clone detector that applies LSTM on ASTs. The CDLH paper [130] does not provide details about their experimental settings. To make a fair comparison, we compare with their reported numbers.

Training. We apply 10-fold cross-validation to train and evaluate DeepSim and the two baseline models on the two datasets. Namely, we partition the dataset into 10 subsets, each time we pick 1 subset as test set, the rest 9 subsets as training set. We repeat this 10 times and each time the picked test subset is different. The reported result is averaged over the 10 times.

For RtvNN, because it needs to extract all the tokens and build vocabulary before training, we have to train and evaluate it using the same full dataset, following the same experiment setting as [131]. We try a range of different super-parameters (e.g., learning rate, layer sizes, regularization rate, dropout rate, various activation functions and weights initializers, etc.) for each model and record their testing errors and F1 scores. For DECKARD, it has no training procedure but a few parameters. We also tune its parameters and choose the set of parameters that achieved the highest F1 score on the full dataset. The optimal super-parameters that achieved the highest F1 score for these models are listed in Table 3.2.

3.4.3 Results on GCJ

3.4.3.1 Recall and Precision

Table 3.3: Results on the GCJ dataset.

Tool	Recall	Precision	F1 score
DECKARD	0.44	0.45	0.44
SdA-base	0.51	0.50	0.50
SdA-unsup	0.56	0.26	0.35
RtvNN	**0.90**	0.20	0.33
DeepSim	0.82	**0.71**	**0.76**

Table 3.3 reports the recall and precision of DeepSim on GCJ compared to the other approaches. Recall means the fraction of similar method pairs retrieved by an approach. Precision means the fraction of retrieved truly similar method pairs in the results reported by an approach. Overall, DeepSim achieved 81.6% recall, while DECKARD 44.0%, RtvNN 90.2%, SdA-

base 51.3% and SdA-unsup 55.6%, and DeepSim achieved much higher precision (71.3%) than DECKARD (44.8%) and RtvNN (19.8%), as well as SdA-base (49.6%) and SdA-unsup (25.6%).

DECKARD achieved low recall and precision. The reason is that to recognize a similar method pair, DECKARD requires the characteristic vectors of parser tree roots for the two methods to be very close, which is essentially the syntactics of the entire code. To better understand this result, we picked all solutions from one competition problem, and found that more than half of the functionally similar code pairs have diverse parser tree structures, making them being predicted into different clusters by DECKARD.

SdA-base and SdA-unsup achieved comparable recall and are better than DECKARD, and SdA-base's precision is much higher than SdA-unsup. DeepSim achieved much higher recall than DECKARD and the two baseline models. This indicates that DeepSim effectively learns higher level features from the semantic matrices than the other approaches, and the encoded semantic matrix is also a better representation than the syntactical representation used by DECKARD.

RtvNN achieved the highest recall, but very low precision. We found that RtvNN almost reports all method pairs as similar. The reason is that RtvNN relies on the tokens and AST to generate the hidden representation for each method and applies a simple distance metric to discriminate them. However, two functionally similar methods may have significant syntactical differences, and two functionally dissimilar methods may share syntactically similar components (e.g., IO operations). After further analyzing the experiment results, we found that the distances between most methods are in the range of [2.0, 2.8]. Through reducing the distance threshold, the precision of RtvNN could be significantly improved (up to 90%), however, its recall also drops quickly (down to less than 10%). As a result, it only achieved F1 score 0.325 at the highest.

3.4.3.2 False positives/false negatives.

The precision of DeepSim is 71.3%, which still has a large improvement space. After checking those false positives and false negatives of DeepSim, we found that this is mainly due to the tool's limitation in handling *method invocations*. For example, for the Problem *Senate Evacuation*, some solution projects use standard loops and assignment statements, while other solution projects

employ utility methods such as *Map* and *Replace*. Because we do not explicitly encode the information inside the callee method into the semantic feature matrices, DeepSim cannot distinguish these utility methods and their corresponding statements. Nevertheless, this is a common limitation for most existing static code similarity measuring techniques [63, 105, 131]. Considering that DeepSim could generate a final hidden representation for each method after training, incorporating this information to encode method invocation instructions is feasible (re-training may be necessary). In addition, utilizing constraints-solving based approaches [69, 76] as a post-filtering to the reported results may also further improve the precision.

3.4.3.3 Time Performance

We also evaluated the time performance of these approaches on the full dataset (in total 1.4M method pairs). We run each tool with the optimal parameters (Table 3.2) on a desktop PC with an Intel i7 4.0GHz 4 cores CPU and GTX 1080 GPU. For DeepSim and the two SdA baseline models, they need to generate semantic feature matrices from bytecode files, so we also include the time of this procedure into the training time. For DECKARD, we use the default maximal number of processes (i.e., 8). We run each tool three times and report the average.

Table 3.4: Time performance on GCJ.

Tool	**Prediction time**	**Training time**
DECKARD	72s	-
SdA-base	1230s	8779s
SdA-unsup	37s	1482s
RtvNN	15s	8200s
DeepSim	34s	13545s

Table 3.4 reports the results. DECKARD does not need training so it has zero training time. SdA-unsup separates latent representation learning and classification phases, thus it takes fewer training epochs to get stable and has the lowest training time among the three models. While

DeepSim has fewer neurons than SdA-base (SdA-base has a huge amount of neurons in the first layer), it takes more computation since its first three layers are applied on each element or row of the semantic feature matrices. Thus, DeepSim takes the longest training time. Although the training phase of DeepSim is slower than the other approaches, it is a one-time offline process. Once the model is trained, it can be reused to measure code similarity.

For prediction, RtvNN takes the least time, as it only needs to calculate the distance between each code pair and compare it with the threshold. DeepSim takes the 2nd least time, even faster than DECKARD, because it only needs to generate the semantic feature matrices and the prediction phase is performed by GPU, which is fast. SdA-unsup takes approximately the same time as DeepSim, while SdA-base is 30X slower, because it has to run the whole model for each code pair.

3.4.4 Results on BigCloneBench

Table 3.5: Results on BigCloneBench.

Tools	**Recall**	**Precision**	**F1 Score**
DECKARD	0.02	0.93	0.03
RtvNN	0.01	0.95	0.01
CDLH	0.74	0.92	0.82
DeepSim	**0.98**	**0.97**	**0.98**

Table 3.6: F1 score for each clone type.

Clone Type	**DECKARD**	**RtvNN**	**CDLH**	**DeepSim**
T1	0.73	1.00	**1.00**	0.99
T2	0.71	0.97	**1.00**	0.99
ST3	0.54	0.60	0.94	**0.99**
MT3	0.21	0.03	0.88	**0.99**
WT3/T4	0.02	0.00	0.81	**0.97**

Tables 3.5-3.6 report the results on BigCloneBench. The recall and precision are calculated according to [124]. The results of DECKARD, RtvNN and CDLH correspond to that reported in [130].

For this dataset, DeepSim significantly outperforms all the other approaches for both recall and precision. The F1 score of DeepSim is 0.98, compared to 0.82 by CDLH. DeepSim does not achieve 1.0 F1 score on T1-ST3 clones (which should be guaranteed by our encoding approaches as discussed in Section 3.2), because the tool misses some data flow when generating DFG from source code, due to the approximation we introduce to handle external dependencies in WALA.

Since the true/false clone pairs from BigCloneBench are from different functionalities, we run another experiment that use all the true/false clone pairs with functionality id 4 as training dataset (since it contains approximately 80% true/false clone pairs of the whole dataset), and the rest data as testing dataset. The result is shown in Table 3.7, which is consistent with the results reported in Tables 3.5-3.6.

Table 3.7: Results of DeepSim trained using data from single functionality.

Clone Type	Recall	Precision	F1 Score
T1	1.00	1.00	1.00
T2	1.00	1.00	1.00
ST3	1.00	1.00	1.00
MT3	0.99	0.99	0.99
WT3/T4	0.99	0.96	0.97

We also note that for WT3/T4 clone type, the F1 score of DeepSim on BigCloneBench is higher than that on the GCJ dataset (this is consistent with the comparison result between BigCloneBench and another OJ dataset, OJClone, as reported in [130]). We inspected several WT3/T4 clone pairs and found that although they are less than 50% similar at the statement level (which is the criterion for WT3/T4 clone type), many of them follow similar code structure and differ only on the sequence of the invoked APIs. In contrast, the GCJ projects are all built from scratch by different

programmers with different code structures. It is hence more difficult to detect functional clones in the GCJ dataset.

3.4.5 Discussion

Why DeepSim outperforms the other DNN approaches. The reasons are three-fold. First, DeepSim is based on the semantic feature matrix that encodes DFG/CFG, which is a higher level abstraction than the lexical/syntactical representation (e.g., source code tokens or AST) used by the other DNN approaches such as RtvNN and CDLH. Compared to the two SdA baselines, DeepSim takes the semantic feature matrix as input, while for the SdA baselines each 88d feature vector $\mathcal{T}$ is manually re-encoded to 8d, and the entire matrix is flattened to a vector. The manual re-encoding may lose information since each element in the 8d vector is actually *categorical data* and a standard neural network model is limited in this scenario. In addition, flattening the entire matrix aggravates the impact of statement re-ordering because the flattened vector is strictly ordered, which causes reporting more false positives.

Second, DeepSim and the two SdA baselines all contain classification layers to support the classification of a method pair, while RtvNN only calculates the distance between the hidden representations of two methods, which may not be applicable as we discussed before. Note that the F1 score of DeepSim, SdA-base and SdA-unsup are all higher than RtvNN, this should be partly attributed to the effectiveness of our binary classification formulation and the fine training stage.

Third, the DeepSim model uses two average pooling layers, which SdA-base and SdA-unsup do not have. The first one computes the average states along all rows of the matrix, which reduces the side effect caused by the statement re-ordering. The second one computes the average states of two different concatenations of final latent representations from a method pair, which guarantees the symmetric property of code similarity, i.e. $S(m_i, m_j) = S(m_j, m_i)$.

Effectiveness of the encoding approach. In the first layer of DeepSim, we map each feature vector $\mathcal{T}$ of size 88 to a 6d vector in the hidden space. To analyze our trained model, we construct eight different feature vectors, each is encoding of a specific operation between variables. We map them into the embedding space and visual them with PCA dimensionality reduction, as shown in

Figure 3.5.

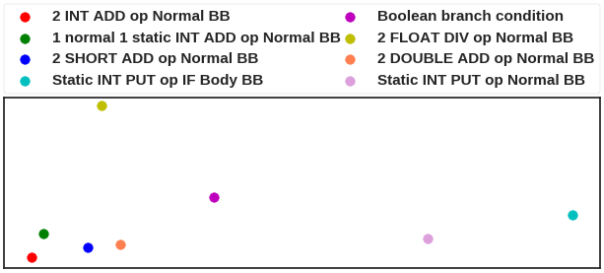


Figure 3.5: First layer hidden representation of 8 different input feature vectors generated by DeepSim.

We can observe that there are four very close points, corresponding to four different instructions: 2 int variables *add* operation in normal basic block, 2 short variables *add* operation in normal basic block, 2 double variables *add* operation in normal basic block, 1 int variable and 1 static int variable *add* operation in normal basic block. According to their meanings, they are quite similar and should be close in the embedding space. This manifests that our encoding does effectively preserve the data flow information, and our model successfully learned the similarity between these operations.

We also note that the two identical operations from different basic blocks (denoted by the plum and cyan points in the figure) are not close in the embedding space. This indicates that the code structure (i.e., control flow) information is also well encoded into the feature vector and successfully learned by our model for measuring functional code similarity.

The visualization of the hidden states for the eight instructions shows the effectiveness of our semantic feature matrices that encode data flow and control flow. More importantly, our specially designed DNN model can learn high-level features from the input semantic feature matrix, and patterns between functionally similar code with different syntactics. This significantly distinguishes our approach from the other approaches, which rely on syntactical representation.

Semantic feature matrix size. In this work, we fix the size of the semantic feature matrix

to 128. It works well for our datasets, but such fixed length reduces computation efficiency for methods with smaller length and may lose some information for methods with larger length. To support variant matrix length, we can integrate our model with recurrent neural network (RNN). However, a challenge is that RNN requires a fixed input sequence order, which is difficult to handle code statement re-ordering. A stacked LSTM with attention mechanism [17] may be a potential solution.

Larger dataset. As a deep learning approach, DeepSim's effectiveness is limited by the size and quality of the training dataset. Building such a large and representative dataset is challenging. Even BigCloneBench only covers 10 functionalities and consists of less than 60K functions, which is a tiny proportion of the full IJaDataset [124]. In future work, this can be addressed by incrementally building a very large dataset through crowd-sourcing platform such as Amazon Mechanical Turk [94]. We also plan to build a web platform so that users can upload samples to try our tool and verify the result. The collected data will help improve the accuracy of our model.

Graph neural network. Recently graph neural networks (GNN) has been proved to be effective in tasks such as graph classification and clustering [73, 139]. Since data-flow and control-flow are both represented in graph, GNN can be a good alternative approach. However, the granularity of the graph (e.g., a node can be a single operand, or an expression) may need careful consideration. And normalization (e.g., normalizing all integer numbers to the same token *INT*) may be still necessary to improve generalization capability of the model.

3.5 Summary

We presented a new approach, DeepSim, for measuring code functional similarity, which consists of a novel semantic representation that encodes the code control flow and data flow information as a compact matrix, and a DNN model that learns latent features from the semantic matrices and performs binary classification. The evaluation of DeepSim on two large datasets, and its comparison with several state-of-the-art approaches show that DeepSim significantly outperforms existing approaches in terms of recall and precision, and meanwhile it achieves very good efficiency. This new approach will benefit existing approaches for searching vulnerable code with existing

CVEs in terms of both accuracy and time performance.

To detect atomicity violations in Rust programs, we first infer which atomic operations could access the same variable from different threads. This requires alias analysis, which determines whether two pointers can point to the same memory, and happens-before relation, which determines whether two operations from different threads can be executed concurrently. Since atomicity violation is essentially semantic bug, we apply heuristics observed from existing research study to check if a set of atomic operations could potentially cause an atomicity violation.

In this chapter, we first show a motivating example, illustrating how MIRAV works at high level. Then we present the technical approach. At last, we present the implementation and benchmark evaluation. We will also discuss the limitations of this approach.

4.1 Motivating Example

```
1  static NUM: AtomicU32 = AtomicU32::new(0);
2
3  fn update() {
4    let mut val = NUM.load(Ordering::Relaxed);
5    val += get_increment(val);
6    // Some other code that use val.
7    do_work(val);
8    NUM.store(val, Ordering::Relaxed);
9  }
10
11 fn main() {
12   let t = thread::spawn(|| {
13     NUM.store(1, Ordering::Relaxed);
14   });
15   update();
16   t.join().unwrap();
17 }
```

Figure 4.1: Atomicity violation on the atomic variable.

We use the code in Figure 4.1 as our motivating example to illustrate our approach, whose MIR is shown in Figure 4.2. For simplicity, we only show part of the MIR. The code executed by the new thread created in main function corresponds to `main::{closure0}` in the IR.

```
1  fn update() -> () {
2    bb0: {
3      _3 = const {alloc1: &AtomicU32};
4      _2 = _3;
5      _1 = AtomicU32::load(move _2, move _4) -> bb1;
6    }
7    bb3: {
8      _1 = move (_7.0: u32);
9      _9 = _1;
10     _8 = do_work(move _9) -> bb4;
11   }
12   bb4: {
13     _12 = const {alloc1: &AtomicU32};
14     _11 = _12;
15     _13 = _1;
16     _10 = AtomicU32::store(move _11, move _13, move _14);
17   }
18 }
19
20 fn main::{closure0}(_1: [closure@src/main.rs]) -> () {
21   bb0: {
22     _4 = const {alloc1: &AtomicU32};
23     _3 = _4;
24     _2 = AtomicU32::store(move _3, const 1_u32, move _5);
25   }
26 }
```

Figure 4.2: MIR for code in Figure 4.1. For simplicity we omit some code.

As explained before, this involves a single atomic variable, `NUM` and three operations: load at line 5 in the MIR, store at line 16 and another store at line 24. More importantly, these three operations are performed concurrently by two threads: function `update` are called in the main thread; the closure function runs in a child thread. Different interleavings thus can be observed in different runs as illustrated below:

```
Interleaving #1
Thread 1:                          Thread 2:
load@line 5
...
                                   store@line 24
store@line 16
```

```
Interleaving #2
Thread 1:                          Thread 2:
                                   store@line 24
load@line 5
...
store@line 16
```

These two interleavings lead to different values stored in `NUM`, and only the latter is correct.

We use the following three components to automatically check if such different interleavings are possible. Note that these components are different from general analyses in that they deal with Rust/MIR language features and are specialized for atomic variables.

1. Alias Analysis: Ensuring that all these atomic operations access the same variable is important, otherwise it is less likely to cause atomicity violation. In the motivating example, all accesses are using the reference obtained from `alloc1`, which is the global variable corresponding to `NUM`. Nonetheless, assume we use the new thread code below, in which the atomic variable being accessed is returned from another function and hence alias information is harder to infer. For this purpose, we implement an inter-procedural alias analysis to compute such information.

```
fn get_num() -> &'static AtomicU32 {
  return &NUM;
}

let t = thread::spawn(|| {
  let num = get_num();
  num.store(1, Ordering::Relaxed);
});
```

2. Happens-Before Analysis: we also need to check that there is no ordering between the atomic operations in different threads to ensure that these accesses can be concurrent. For example, in the code below, where `t.join()` is called before the `update` function, only one interleaving is possible and hence no atomicity violation. We use static happens-before analysis to reason about such ordering constraints.

```
fn main() {
  let t = thread::spawn(|| {
    NUM.store(1, Ordering::Relaxed);
  });
  t.join().unwrap();
  update();
}
```

3. Pattern Detection: To detect the potential causes for atomicity violations, we first collect all atomic operations in an inter-procedural manner (line 5, 16 and 24 in the MIR), as well as all thread events (e.g., fork, join) as required for checking ordering. We then match them with patterns discussed in Tables 1.1 and 1.2. To achieve this, we run a whole program analysis pass. Then we select potential atomic operation pairs from each thread, and check against atomic operation in other threads to determine whether there is a potential atomicity violation. This step reuses the result computed by alias and happens-before analysis as well as heuristics summarized from

existing research study and real-world bugs. In this simple example, line 24 in the MIR can run between line 5 and line 16 from a different thread, matching the last pattern in Table 1.1. Therefore, we report an atomicity violation.

4.2 Technical Approach

In this section, we will first give a brief introduction of the Rust mid-level intermediate representation, MIR. Then we present our alias analysis for atomic variables and static happens-before relations for handling thread fork and join events. With these analysis results, we present the algorithm for detecting atomicity violations.

4.2.1 MIR

Rust source code is translated to target code through the pipeline AST $\rightarrow$ HIR (high-level IR) $\rightarrow$ MIR $\rightarrow$ LLVM (or cranelift) $\rightarrow$ target code. MIR has several key characteristics that make it suitable for our analysis: It is based on control-flow graph, has no nested expressions and uses explicit types. The MIR representing a single function consists of declarations (user-declared bindings and temporaries) and basic blocks. A basic block consists of a set of statements and a terminator. Below we give a brief about the statements and terminator. Note that we omit some types of statements and terminators that are not important to our analysis. For full grammar of MIR, readers can refer to [12].

⟨*basic_block*⟩ ::= ⟨*stmt*⟩* ⟨*terminator*⟩

⟨*stmt*⟩ ::= ⟨*lvalue*⟩ = ⟨*rvalue*⟩

⟨*terminator*⟩ ::= goto(⟨*bb*⟩) // ⟨*bb*⟩ is index to a basic block

| panic(⟨*bb*⟩)

| if(⟨*lvalue*⟩, ⟨*bb*⟩, ⟨*bb*⟩)

| switch(⟨*lvalue*⟩, ⟨*bb*⟩*)

| ⟨*lvalue*⟩ = ⟨*lvalue*⟩(⟨*lvalue*⟩*)

| ⟨*return*⟩

⟨*lvalue*⟩ ::= B // reference to a user-declared binding

| TEMP // a temporary introduced by the compiler

| ARG // a formal argument of the fn

| STATIC // a reference to a static or static mut

| ⟨*lvalue*⟩.f

| *⟨*lvalue*⟩

⟨*rvalue*⟩ ::= use(⟨*lvalue*⟩) // just read an lvalue

| &'region ⟨*lvalue*⟩

| &'region mut ⟨*lvalue*⟩

| ⟨*lvalue*⟩ as ⟨*type*⟩ // type casting

| CONSTANT

Figure 4.2 shows a MIR example. Global variables are accessed through references to new wrapper variables (usually with prefix `alloc`), which preserves size and alignment information about the original global variable. And struct member functions are normalized to global functions (the reference to the struct instance becomes the first parameter of the normalized function). Accesses to struct fields are normalized too, using the index of the corresponding field. For example,

in line 8 of the MIR, `(_7.0: u32)` means the first field of the object represented by `_7`.

4.2.2 Alias Analysis

In order to infer whether two atomic operations access the same variable, alias analysis is desired. Our alias analysis has three special characteristics.

First, we do not model heap (i.e., the dynamically allocated object), since we are not interested in which object that a reference/pointer points to. Existing work usually formalizes it as a CFL-reachibility problem [141]. In our prototype, we use datalog since any CFL-reachibility problem can be converted to a datalog program [116].

Second, field accesses are explicit reference/assignment in MIR (as shown in Section 4.2.1), instead of pointer arithmetic or memory load/store as in LLVM, so we handle them in a way similar to handling normal reference assignment. Specifically, for field access `a = b.f`, we treat the left value `a` and right value `b.f` as alias, meanwhile, we will collect the fact that `b` is a prefix of `b.f` (i.e., `b.f` is reachable from `b` as in CFL-reachibility). Statement `b.f = a` can be handled in a similar way. To determine whether a pair of field references `a.f1.f2` and `b.f3.f4` is alias, we will transitively check if their prefixes are alias and the field are the same (i.e., checking {`a.f1`, `b.f3`} first, then {`a`, `b`}). In addition, we will also leverage the explicit types provided in MIR for type filtering to improve the precision.

Third, our alias analysis is sparse: it only analyzes a very limited subset of all the pointers/references in the program. They are, namely, callable types (e.g., functions, closures, etc.) and atomic types. Alias information for callable type is used for building call graph and collecting thread events; alias information for atomic type is used for collecting all atomic operations. This helps to scale MIRAV on large programs. The datalog input facts and rules are listed in Figure 4.3.

Note that our alias analysis is not intended to be precise and complete, since we skip assignments that do not involve atomic types and callable types, and it is flow-insensitive and context-insensitive.

Given a target Rust project, we need to traverse the MIR of every function in the source code and its dependency (including Rust standard library), in order to collect all the necessary facts in

```
// Input facts
assign(x, y).
prefix(x2, x).
call_graph(x, m).
// n is the index of the actual/formal argument
actual_arg(stmt, n, x).
formal_param(m, n, x).
actual_ret(stmt, x).
formal_ret(m, x).
// Corresponding to de-reference in MIR
load(x, p).
store(p, x).

// Rules
alias(x, y) :- assign(x, y).
alias(x, y) :- alias(x, z), alias(z, y).
alias(x, y) :- alias(x2, y2), prefix(x2, x), prefix(y2, y),
               types(y, t), types(x, t).
alias(x,y) :- load(x, p), store(p, y), types(x, t),
              types(y, t).
call_graph(x, m) :- alias(x, y), call_graph(y, m).
alias(x, y) :- call_graph(stmt, m), actual_arg(stmt, n, y),
               formal_param(m, n, x).
alias(x, y) :- call_graph(stmt, m), actual_ret(stmt, x),
               formal_ret(m, y).
```

Figure 4.3: Datalog rules for alias analysis.

Figure 4.3.

4.2.3 Static Happens-Before Relations

The atomicity violation described in this work involves at least two threads [1]. The thread start events and join events, together with the sequence order in each thread, impose a partial order $\rightarrow$ on the whole programs. We reuse the formal definition from [77] (an event in the context of this work is either thread start/join or an atomic operation):

- Given two events e_1 and e_2 in the same thread, $e_1 \rightarrow e_2$ if e_1 precedes e_2 in program sequence order.

[1] For single-threaded program, interrupt handler in kernel space may also cause atomicity violation, which is out of our scope.

- Given an event e_1 in thread t_1, thread start event e_2 in thread t_1 and another event e_3 in the started thread t_2, $e_1 \rightarrow e_2 \rightarrow e_3$ if e_1 precedes e_2 in program sequence order.
- Similarly, given an event e_1 in thread t_1, thread join event e_2 in thread t_1 and another event e_3 in the thread t_2 which is being joined, $e_3 \rightarrow e_2 \rightarrow e_1$ if e_2 precedes e_1 in program sequence order.

And this partial order is transitive, irreflexive and antisymmetric:

- $\forall e_1, e_2, e_3$, if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$, then $e_1 \rightarrow e_3$.
- $\forall e$, $e \not\rightarrow e$, which means any event cannot happen before itself.
- $\forall e_1, e_2$ where $e_1 \neq e_2$, if $e_1 \rightarrow e_2$, then $e_2 \not\rightarrow e_1$.

With happens-before relation, we can check if a specific interleaving of a set of atomic operations are valid. For example, thread t_1 has two atomic operations e_1 and e_2, while thread t_2 has another atomic operation e_3, and all the three operations access the same variable. If $e_1 \rightarrow e_2$, $e_3 \rightarrow e_1$ (or $e_2 \rightarrow e_3$), then there is only one viable interleaving ($\{e_3, e_1, e_2\}$ or $\{e_1, e_2, e_3\}$, depending on the pre-conditions) and hence no potential atomicity violation. This can help reduce false positives significantly.

Note that memory ordering of atomic operations may affect the execution order in a single thread, e.g., two atomic operations on different variables with *relaxed* memory ordering can be exchanged in a single thread. In this work, we do not fully model all different memory orderings. But we will consider the *relaxed* ordering when checking if a set of atomic operations could cause atomicity violation, see Section 4.2.4 below.

In order to support effectively checking the happens-before relation of any two events, we build a directed graph to connect all events on the fly when traversing the whole program [82]. The direction of the edge represents the happens-before relation of the two events connected by the edge. For an arbitrary pair of events e_1 and e_2, we infer $e_1 < e_2$ if the node of e_2 is reachable from e_1.

In addition to thread events, lock events can impose more constraints on possible interleavings of a multi-threaded program. Many existing work combine lockset with happens-before to detect concurrency issues [107, 93]. In this work, we do not consider lock events. Through initial study of a set of real world Rust concurrency programs that uses atomic types, we found that in most cases atomic operations are not guarded by any locks. This is consistent to the convention that developers usually use atomic types to avoid expensive locks.

4.2.4 Detecting Atomicity Violations

With the alias result, we run an analysis pass on the MIR of the entry point function (i.e., main). The analysis will traverse every reachable function following the call graph constructed with the alias result. Each time when a thread event or atomic operation is visited, we add it into the happens-before graph, and preserve the atomic operations per thread. After the traversing is complete (and the intact happens-before graph is constructed), we start to run the detection algorithm, as shown in Algorithm 1.

First, we iterate all collected operations for each thread (line 4), and each time we pick an operation *op1*, we try to get a next operation *op2* (line 7), check if *is_pair* is satisfied (line 8). If it is, we preserve this pair, and repeat this process, otherwise, we continue to check the next operation (line 11).

The function *is_pair* applies several rules to determine whether *op1* and *op2* can be treated as a pair of operations:

- *op1* and *op2* are alias (using the alias result in Section 4.2.2)
- *op1* is a load and *op2* is a store, and the new value being stored depends on the value loaded from *op1* (i.e., flow dependency).

Next, we iterate all collected pairs for each thread (line 15). Each time we pick a pair, and try to get an operation *op3* from a different thread (line 17-18). The three operations should satisfy several preconditions:

Algorithm 1: Detecting Atomicity Violation.

```
   Data: atomic_map: map<tid, vec<atomic_op> >
 1 Function detectAV (atomic_map)
 2     all_pairs := map<tid, pair<atomic_op> >;
 3     avs := vec<triple<atomic_op> >;
 4     forall (tid, atomic_ops) in atomic_map do
 5         forall op1 in atomic_ops do
 6             cur_op = op1;
 7             while op2 = get_next_op(cur_op, atomic_ops) do
 8                 if is_pair(op1.var, op2.var) then
 9                     all_pairs[tid].push(op1, op2);
10                     break;
11                 cur_op = op2;
12             end
13         end
14     end
15     forall (tid, pairs) in all_pairs do
16         forall (op1, op2) in pairs do
17             other_ops = ops_in_other_threads(tid);
18             forall op3 in ops do
19                 if !alias(op3.var, op1.var) then
20                     continue;
21                 if happens_before(op3, op1 || happens_before(op2, op3) then
22                     continue;
23                 if op3.is_store() then
24                     add_report(avs, op1, op2, op3);
25                 else if is_suspicious_region(op1, op2) then
26                     add_report(avs, op1, op2, op3);
27             end
28         end
29     end
30     post_processing(avs);
31 end
```

- *op3* must be alias of *op1* (line 19), which means that they are (likely) accessing the same variable.

- There is no happens before relation $op3 \rightarrow op1$ or $op2 \rightarrow op3$, which means there could be more than one interleaving.

With all these satisfied, we check if *op3* is a store operation (line 23). If it is, the three operations form one pattern studied in Table 1.1, and we add it to the report list (line 24). Otherwise, we further check if the two operations *op1* and *op2* and the code region between them is suspicious (i.e., intended to be atomic). If it is, we also add it to the report list.

Function *is_suspicious_region* checks if the code region between statement *op1* and *op2* is (or is included in) a potential atomic code region with heuristics observed from existing research study [98, 137]:

- If the region contains any unsafe code (e.g., unsafe memory operations, calling foreign functions, mostly *libc*), or the value loaded from *op1* is used later in unsafe code, we consider that it should not be interleaved.

- If the region contains atomic operations on other variables, and the memory ordering is *relaxed*, we consider that it should be atomic.

- If *op1* is load and *op2* is store, and there is a flow dependency between them, we consider that it should be atomic.

We stress that atomicity violation is in fact a kind of semantic bugs that depend on developers' intention. Therefore, such heuristics are an important way to infer the code underlying assumption, with false positives allowed.

After collecting all the reported atomicity violations, the algorithm runs post-processing (line 30). This step removes duplicate atomicity violations caused by two threads calling the same code. It also merges the atomicity violations that are reported on the same set of atomic operations in the same function pairs, in order to reduce user's efforts on verifying them.

4.3 Implementation

We have implemented a prototype tool, MIRAV, as a cargo subcommand, for Rust 1.59 nightly version. We use the *petgraph* package for the static happens-before graph. For effectiveness, we use the *datafrog* package for running the datalog rules (Figure 4.3) on input facts in memory, instead of exporting them to disk for external tools to process (e.g., Souffle).

When collecting input facts, ideally we need to traverse the MIR of every function in the target project and all its dependent crates, including the large Rust standard library. However, early experiments showed that the tool visited a huge amount of functions and the datalog took long time (>1000s on a small two-threaded program) to reach a fixed point. To address it, we first collect all constants that are (transitively) reachable from the HIR of the current project. For each constant that represents a function symbol, we traverse the MIR of the corresponding function.

In addition, when visiting statements in the MIR for generating input facts, we skip any operands that are (references/pointers of) primitive types (e.g., int, float, char, etc.) and user-defined types that do not contain any field with atomic types. Note that this type checking is transitive to avoid missing atomic variable. For example, given user type `struct {a: UserTy, }`, we need to further check if the type of field `a`, `UserTy`, contains any field with atomic type.

To collect thread events, MIRAV currently only identifies APIs provided in the *std::thread* library. We model the semantic of each API and add the new event into the static happens-before graph (e.g., `thread::spawn` is modelled as start event). For atomic APIs that are both load and store (e.g., *compare_exchange*), we model it as a load operation and a store operation (but we will not select the two operations as a pair in Algorithm 1).

For practical purpose, we also add filtering into MIRAV. Through initial experiments, we found that a few false positives are reported inside Rust standard library, which is caused by either low-level synchronization primitives or our imprecise heuristics. Therefore, we filter those out when reporting analysis results.

Table 4.1: Benchmarks. The bug fix commit ID is used as the bug ID.

Benchmark	Description	Bug ID	#TP/#Total
rand	The de-facto Rust library for random number generation.	e0e8263	1/1
		ae3a416	1/1
crossbeam-epoch	One component in the crossbeam project, which is one of the most popular Rust concurrency libraries. It provides epoch-based garbage collection for building concurrent data structures.	268c028	1/2
crossbeam-channel	A library that provides multi-producer multi-consumer channels for message passing.	9d42394	1/4
		1fbf84b	2/2
sled	A high-performance embedded database written in Rust.	21a3159	1/12
		07a3ccc	0/17

4.4 Evaluation

We evaluate our prototype tool on both open-source projects with known atomicity violations and wild projects. In this section, we first introduce our benchmark setting and analyze the results. Then we discuss initial results on scanning wild projects.

4.4.1 Benchmark

To evaluate the effectiveness of this tool, we have collected a set of real atomicity violations from popular Rust projects on Github and existing research study [137], as shown in Table 4.1.

4.4.2 Results

We run our tool on each project (specifically, on the parent commit of the bug fixing commit). For projects that are pure libraries (e.g., rand, crossbeam-epoch), we use their examples code as the analysis entry point. If the entry point function runs only one thread, we will create a new thread which runs the same code.

The results are shown in Table 4.1, MIRAV successfully detected 6 out of the 7 atomicity violations (on *crossbeam-channel-1fbf84b* it reports the same atomicity violation from two example code). Below we will discuss them in details.

The atomicity violation fixed in *rand-e0e8263* is shown in Figure 4.4. There are two atomic

```
1  fn is_getrandom_available() -> bool {
2    static GETRANDOM_CHECKED: AtomicBool = ATOMIC_BOOL_INIT;
3    static GETRANDOM_AVAILABLE: AtomicBool = ATOMIC_BOOL_INIT;
4
5    if !GETRANDOM_CHECKED.load(Ordering::Relaxed) {
6      let mut buf: [u8; 0] = [];
7      let result = getrandom(&mut buf);
8      let available = if result == -1 {
9        let err = io::Error::last_os_error().raw_os_error();
10       err != Some(libc::ENOSYS)
11     } else {
12       true
13     };
14     GETRANDOM_AVAILABLE.store(available, Ordering::Relaxed);
15     GETRANDOM_CHECKED.store(true, Ordering::Relaxed);
16     available
17   } else {
18     GETRANDOM_AVAILABLE.load(Ordering::Relaxed)
19   }
20 }
```

Figure 4.4: Root cause of *rand-e0e8263*.

variables involved, given that the ordering are all `Relaxed`, the below execution scheduling is possible:

1. Thread 1 reaches line 5, since `GETRANDOM_CHECKED` is initialized as false, it enters the if body and runs line 15, while line 14 is not executed yet.

2. Thread 2 enters the function and reaches line 5, now the value is *true* (updated by thread 1), so it jumps to line 19 and load an value of `GETRANDOM_AVAILABLE` that is not correctly set by thread 1 yet.

3. Thread 1 continues the execution and updates `GETRANDOM_AVAILABLE`.

In this case, the atomicity violation is not on the variable `GETRANDOM_AVAILABLE` itself. Instead, the entire if body (line 7 - line 15) should be atomic. Since this code region contains both unsafe code and an atomic operation with *relaxed* ordering, MIRAV is able to detect it. The other case, *rand-ae3a416* has very similar root cause, we omit its discussion.

```
1  pub fn insert(&self, t: Bag){
2    let n = Box::into_raw(Box::new(
3      Node { data: t, next: AtomicPtr::new(ptr::null_mut()) }));
4    loop {
5      let head = self.head.load(Relaxed);
6      unsafe { (*n).next.store(head, Relaxed) };
7      if self.head.compare_and_swap(head, n, Release) == head { break }
8    }
9  }
10
11 pub unsafe fn collect(&self) {
12   let mut head = self.head.load(Relaxed);
13   self.head.store(ptr::null_mut(), Relaxed);
14
15   while head != ptr::null_mut() {
16     let mut n = Box::from_raw(head);
17     n.data.collect();
18     head = n.next.load(Relaxed);
19   }
20 }
```

Figure 4.5: Root cause of *crossbeam-epoch-268c028*.

The atomicity violation on *crossbeam-epoch* involves accesses to the same atomic variable in different functions. The root cause code is shown in Figure 4.5. In function `insert`, `self.head` could be updated at line 7. In function `collect`, `self.head` is first read to `head`, then set to be `null`. However, if `insert` is executed by another thread between line 12 and line 13, the local `head` becomes a stale value and the following while loop becomes incorrect. In addition to this real atomicity violation, MIRAV reports one more with the load/store pair in function `insert`. This is in fact a false positive since the *compare_and_swap* at line 7 will fail if another thread updates `self.head` in function `collect`.

The atomicity violation on *crossbeam-channel-9d42394* is very typical. The root cause code is shown in Figure 4.6. The value of `self.tail.index` is first read to local variable `tail`, after a few statements, `self.tail.index` could possibly be updated to value `new_tail` (which is computed from `tail`) at line 8 through `compare_exchange_weak`. Finally, at line 17, `self.tail.index` is updated again to value `new_index`, which depends on `new_tail`.

```
1  fn start_send(&self, token: &mut Token) -> bool {
2    let mut tail = self.tail.index.load(Ordering::Acquire);
3    // ... omitted code
4
5    let new_tail = tail + (1 << SHIFT);
6
7    // Try advancing the tail forward.
8    match self.tail.index.compare_exchange_weak(
9        tail, new_tail, Ordering::SeqCst, Ordering::Acquire) {
10     Ok(_) => unsafe {
11       // If we've reached the end of the block, install the next one.
12       if offset + 1 == BLOCK_CAP {
13         let next_block = Box::into_raw(next_block.unwrap());
14         let next_index = new_tail.wrapping_add(1 << SHIFT);
15
16         self.tail.block.store(next_block, Ordering::Release);
17         self.tail.index.store(next_index, Ordering::Release);
18         (*block).next.store(next_block, Ordering::Release);
19       }
20
21       token.list.block = block as *const u8;
22       token.list.offset = offset;
23       return true;
24     }
25     Err(t) => { // omitted code }
26   }
27 }
```

Figure 4.6: Root cause of *crossbeam-channel-9d42394*.

If another thread updates the same atomic variable when the current thread runs between line 9 and line 17, the value of `next_index` becomes incorrect. The complete code of this function is complicated, involving multiple atomic variables and operations, so MIRAV reported 3 false positives (1 caused by the `compare_exchange_weak`, and 2 are on different atomic variables) in addition to the correct one.

crossbeam-channel-1fbf84b is similar to *rand-e0e8263* (two runnable examples report the same atomicity violation), and *sled-21a3159* is similar to *crossbeam-channel-9d42394*. The only difference is that in *sled-21a3159* we model the read/write of *RwLock* (together with the drop of the corresponding guard) in the same way as load/store of atomic types. We also observed that *sled-*

21a3159 reports much more false positives than the previous programs. We analyzed these false positives and found that most are due to the heuristic patterns we used (see Section 4.2.4).

```
1  fn allocate_inner<'g>(&self, new: Update, guard: &'g Guard,
2      ) -> Result<(PageId, PageView<'g>)> {
3    // ... omitted code
4    let pid = self.next_pid_to_allocate.fetch_add(1, Relaxed);
5    trace!("allocating pid {} for the first time", pid);
6    let new_page = Page { update: None, cache_infos: Vec::default() };
7    let page_view = self.inner.insert(pid, new_page, guard);
8
9    (pid, page_view)
10 }
```

Figure 4.7: Root cause of *sled-07a3ccc*.

The atomicity violation fixed in *sled-07a3ccc* is very interesting. And our tool failed to discover it. The reason is that this bug is related to a high level invariant in the program. As shown in Figure 4.7, the function `allocate_inner` is responsible for allocating a new page and returning it to the caller. Each page is assigned an index, which is from the atomic variable `next_pid_to_allocate`. If we inspect only this function, there is no atomicity violation. However, the program itself assumes an invariant: the allocated pages should be contiguous (i.e., the index values of allocated pages are in ascending order). This is important when recovering pages which are persisted to disks monotonically. In the buggy code, since it uses atomic operation instead of lock, the below scheduling is possible:

1. Thread 1 enters this function and runs line 4 first (assume the value of `pid` is 1), then gets switched out.

2. Thread 2 enters this function, runs line 4 too, and returns the corresponding `pid` (which should be 2) and page.

3. Thread 1 continues and returns the corresponding `pid` (with value 1) and page.

This apparently breaks the assumed invariant. Essentially, the programmers assumed an atomicity in this function which can not be guaranteed by atomic types: the code execution between line 4 and line 9 should not be interleaved by another update to `next_pid_to_allocate` in another thread.

Since our tool expects at least two atomic operations from two statements for one thread and another atomic operation from another thread (see Algorithm 1), it cannot detect this bug. In order to address this kind of atomicity violations, we consider that a reasonable solution is to have the programmers mark special code regions to allow more conservative analysis, without introducing lots of false positives.

From Table 4.1, we can also note that MIRAV reports reasonable false positives. In *rand*, there is no false positives since it is a relatively small project and there are not too much atomic types used. In *crossbeam*, though it reports a few false positives, the code sites of the false positives are close to the true atomicity violation, hence are still valuable to users. In *sled*, it reports lots of false positives. We consider it acceptable given the large code base size of this project as well as its complicated program logics.

4.4.3 Real-world vulnerabilities

We also use MIRAV to scan wild project in order to find new atomicity violations. In total, we collect 25 Rust projects from Github that contain code using atomic types. All of them are libraries, and many do not include executable examples or testing cases. To automate the tool, we choose to create fake entry points for them, in which we create two threads, each calls a public function in the library that contains atomic operations. We will create multiple such entry points if the target project has multiple publicly visible functions that contain atomic operations. We analyze the generated reports and discover one potential atomicity violation. We have submitted an issue for it and are waiting for the maintainer to confirm it. For security concern, we only present the simplified code and our analysis below.

The code snippet in Figure 4.8 shows a simplified vulnerability detected by MIRAV. The crate implements a thread-safe single-producer-multiple-consumer circular queue, in which, producer

```
pub fn publish(&mut self, val: &T) {
  let widx = self.write_idx.fetch_add(1, Ordering::SeqCst);
  // update buffer

  // ... omitted code

  if self.full() {
    self.read_idx.store(new_ridx, Ordering::SeqCst);
  }
}

pub fn read_next(&self, token: &mut ReadToken) -> T {
  let oldest = self.read_idx.load(Ordering::SeqCst);
  // ...omitted code

  let next_write = self.write_idx.load(Ordering::SeqCst);
  let ridx = if wgap >= self.buf_len { oldest } else { .. };

  // ...omitted code
}
```

Figure 4.8: Potential atomicity violation found by MIRAV.

invokes `publish` to insert element into the queue and consumer uses `read_next` to obtain a element in the queue in chronological order. However, the pair of store (`fetch_add` and `read_idx.store`) could be interrupted by the load operation (`read_idx.load`) in `read_next` function invoked by another thread. In this case, the index pointing to the oldest element in the queue are no more consistent between producer and consumers, which could lead to logical errors as loaded data sequences by producers might not be in chronological order.

4.5 Limitations and Discussions

Detecting atomicity violation is a highly challenging problem, since it is related to the semantics of the program. Our approach is limited in several aspects. First, we use flow-insensitive and context-insensitive alias analysis, which is imprecise and may result in either incorrect call graph edge or incorrect alias relationship between two references/pointers. Second, we leverage heuristics when determining whether three atomic operations from two threads could cause poten-

tial atomicity violation, which is neither sound nor precise. Nonetheless, it helps discover bugs in real-world projects. Third, memory ordering of atomic operations is not fully considered, which may cause false positives or missing true positives.

The prototype implementation also has limitations. First, we currently only handle atomic types and thread APIs provided by the Rust standard library. But some projects may use custom atomic types and thread APIs from third-party libraries (e.g., *parking-lot*). The tool needs to support all such popular libraries in order to run on more real-world Rust projects. Second, the tool does not support asynchronous Rust programs. Many recent Rust projects prefer asynchronous programming model with multi-threading support (by using the *tokio* library). The tool needs to support it when collecting input facts and also computing alias.

4.6 Summary

We present MIRAV for detecting atomicity violation bugs in Rust projects. Rust claims memory safety and race-free in safe code, but atomicity violations can still occur in programs written in only safe code. MIRAV runs on MIR generated from the source code, since MIR is CFG-based and has explicit types. It first performs alias analysis on atomic variables to infer the alias relationship between variables accessed by different atomic operations. It then runs whole program analysis to collect atomic operations per-thread and check if the atomic operations from different threads match specific patterns and could potentially cause atomicity violation. The implemented prototype is evaluated on a collection of real atomicity violation bugs as well as wild Rust projects from Github. The results indicate the effectiveness of this approach.

5. SECURING ORIGIN SENSITIVE DATA-FLOW

In this chapter, we present our approach for enforcing statically computed program properties at runtime. Specifically, we compute the static data-flow of the target program. By introducing the concept of origin, we can identify valid cross-origin data-flow at compile time. Then we instrument the program and, with the help of our origin-based heap allocator, efficiently check any invalid cross-origin data-flow at runtime.

We will show a running example that illustrates how our approach works. Then we present the technical details, including computing origin data flow, the origin-based heap allocator and the instrumentation. At last, we present the implementation details and the evaluation on a real-world benchmark. The limitations will also be discussed.

5.1 Running Example

We use an example simplified from HeartBleed in Figure 5.1 to illustrate DFO. The program first reads configuration through the function `initServerConfig` to `config`, which contains server confidential data. The function `doWork` has a loop that keeps reading data and handles it. Since in function `getMalData` the specified length is larger than the actual `data` array, the call to `memcpy` in `heartbeat` will have out-of-bound read, causing data to be leaked in function `print`. This is exactly the root cause of the infamous HeartBleed vulnerability.

To prevent HeartBleed at runtime, one solution (e.g., Softbound) is to track and propagate the memory boundary information (optionally with support of shadow memory) for each pointer. When `memcpy` is invoked, the boundary of `ssl->data` is retrieved, hence the overread is detected. Another solution (e.g., DFI) is computing a static data-flow graph and enforcing it at runtime. Specifically, the reaching definition set for every memory location is statically computed. At runtime, for each write, it updates the reaching definition for the accessed memory location. For each read, it compares the stored runtime reaching definition with the statically computed set. If it is not in that set, an error will be reported.

```
1  Config *config = NULL;
2  typedef struct {
3    void *data;
4    unsigned length;
5  } SSL;
6
7  void heartbeat(SSL *ssl) {
8    uint8_t *buffer = (uint8_t*)malloc(ssl->length);
9    // Heap buffer overread
10   memcpy(buffer, ssl->data, ssl->length);
11   print(buffer, ssl->length);
12   free(buffer);
13 }
14
15 SSL getMalData() {
16   SSL ssl;
17   ssl.data = malloc(512);
18   // ... Fill in data.
19   ssl.length = 65535; // Malicious length
20   return ssl;
21 }
22
23 // origin_entry
24 void handshake(Config *config) {
25   read(config);
26   SSL ssl = getMalData();
27   heartbeat(&ssl);
28   free_ssl(&ssl);
29 }
30
31 void doWork(Config *config) {
32   // This loop can be in parallel.
33   for (;;) handshake(config);
34 }
35
36 int main() {
37   // Read server confidential data
38   config = initServerConfig();
39   doWork(config);
40   return 0;
41 }
```

Figure 5.1: A running example.

Both solutions have high runtime overhead, discouraging them to be deployed in production. Moreover, DFI cannot distinguish the memory written by the same instruction in different contexts. For example, a real-world version of the function `doWork` may run each iteration in different threads. The read to `ssl->data` in `heartbeat` could read an out-of-bound memory region that is allocated by `getMalData()` of another thread. And this read does not violate the runtime check against the statically computed reaching definition set. In case of single-threaded programs, utility functions for memory manipulation called at multiple different sites will result in an over-approximated reaching definition set, causing DFI to fail to prevent invalid accesses too.

From the perspective of attackers, the overread in the above example is only meaningful if other users' data or the server confidential data is leaked. This leads to our key insight: efficiently enforcing valid data-flow across different **origins**.

An origin is an abstraction of a proportion of program execution. It contains all the instructions inside a specified entry point, along with all the data accessed by them. Instructions in an origin may read/write data allocated by another origin. We call it **cross-origin data-flow**. For example, we can treat `handshake` as an origin, in which `ssl->data` in the same iteration (and possibly some fields of `config`) can be read but not write. `ssl->data` allocated by another iteration in another thread is not allowed to be accessed. Similarly, write to `config` is not allowed.

With this abstraction, the problem becomes how to efficiently and effectively enforce valid cross-origin data-flow. In this work, we use an origin-sensitive analysis to statically compute the valid cross-origin data-flow, and instrument the program to ensure the computed data-flow. We also design a custom heap allocator to reduce the number of checks at run time. For example, assume `handshake` is specified as an origin, all heap memory it allocates will be put in a different region. The overread occurred in `heartbeat` cannot touch the memory of `config`. More importantly, `handshake` in different iterations are different origins, which means the overread can never touch the data in a different iteration.

Figure 5.2 shows an informal instrumentation for the code example above, with inserted code colored in gray (the real instrumentation is performed on IR instead of raw source code). The

```
1  __thread uint16_t __rt_origin_id;
2
3  void heartbeat(SSL *ssl) {
4    uint8_t *buffer = (uint8_t*) __xmalloc(ssl->length, __rt_origin_id);
5     __check_read(ssl->data);
6     __check_write(buffer);
7    // Heap buffer overread
8    memcpy(buffer, ssl->data, ssl->length);
9    // ...
10 }
11
12 SSL getMalData() {
13   SSL ssl;
14   ssl.data = __xmalloc(512, __rt_origin_id);
15   // ...
16 }
17
18 // origin_entry
19 void handshake(Config *config) {
20   __rt_origin_id = __get_origin_id();
21   __check_read(config);
22   read(config);
23   // ...
24    __restore(__rt_origin_id);
25 }
```

Figure 5.2: Instrumentation for the running example.

thread local variable `__rt_origin_id` maintains an ID for the current origin. For example, since `handshake` is defined as an origin entry, `__rt_origin_id` is updated at the beginning of it (and restored at the end of it). Allocation calls are all replaced by the calls to our customized heap allocator, which will allocate objects of different origins into different regions in the heap. For each read/write, a checking function call `__check_read`/`__check_write` is inserted immediately before it. For example, for the `memcpy` in function `heartbeat`, two such checking calls are inserted. At runtime, the checking function will query the actual origin ID of the memory being accessed, then check if the runtime origin has the right access permission to it. If it does not, an invalid cross-origin data-flow is reported and the program will be terminated.

5.2 Technical Approach

DFO has three phases:

1. Identify origin entries in the target program and perform static analysis to compute the valid cross-origin data-flow.

2. Instrument the program to use our origin-based heap allocator and to add runtime checks with respect to the statically computed cross-origin data-flow.

3. Run the instrumented program along with the DFO runtime. Once DFO detects any unintended cross-origin data-flow, a runtime error will be raised.

We stress that the scope of DFO is not to detect and to prevent all invalid data-flow, instead, we allow any intra-origin data-flow and only report *unintended cross-origin data-flow*, since our key observation is that to launch a data-only attack, there will normally be illegal data-flow between logically isolated components of a program, i.e., different origins. Nevertheless, our approach can be combined with other approaches to protect intra-origin data-flow, as discussed in Section 5.5.

5.2.1 Identifying Origin Entries

We use the concept of **origin** to abstract a proportion of program execution and all data accessed/owned by it. Given a real target program, we need to concretize this abstraction. Through studying a collection of recent CVEs, we identify several effective concrete definitions of origins: *thread*, *event handler*, and *user input*. Each origin has an entry point and an origin ID. For example, if we define origin as thread for a multi-threaded program, the entry point for each origin is the callback function of the corresponding thread. Note that users can also define origin as a normal program execution starting from a specified function (which becomes the entry point of the defined origin).

Origin entry point function can be called in a loop. In the example below, `t_func` is the entry point, which will be called 10 times by the loop. In order to distinguish the program execution of the `t_func` in different iterations, we propose **static origin** and **dynamic origin**.

```
for (int i = 0; i < 10; ++i) {
    std::thread t(t_func);
}
```

We assign a fixed ID to static origin, which will be used by all instructions and memory objects owned by it. For dynamic origin, at the analysis phase, we assign a special ID to it. At runtime, we assign a unique value to it every time the program execution enters it, by increasing an atomic global counter.

Origin entries can be inferred automatically from the source code, annotated or configured by the developer, or via a combination of them. In this work, we rely on manually added C/C++ custom attributes to annotate origin entries, with the format of:

```
__attribute__((annotate("O[<id>]")))
void foo(...) {}
```

The `<id>` is the origin ID specified by users. We reserve number *1* for default origin starting from the `main` function. Number *0* is for dynamic origin. In the example above, `handshake` can be specified as an origin entry with ID *0*.

5.2.2 Origin Data Sharing Graph

With identified origin entries, we can extract an origin data sharing graph (ODSG) from the program, which captures the valid *data sharing* among different origins, i.e., which origin is allowed to read or write which heap object. A heap object can be shared by multiple origins. The ODSG provides detailed information on *how* each heap object is shared across the origins.

In Algorithm 2, we first compute the set of instructions can be reached by each origin entry (line 4-8). Then for each instruction, we compute the points-to set of its operand (line 16), and we assign the set of origins that own this instruction and corresponding permissions to each memory object in the set (line 17-21). Note that `Permission` is a pair of origin ID and read/write permission. We omit the read/write permission later when discussing creating new origins for shared memory objects.

Algorithm 2: Computing ODSG.

```
   Data: Origin entries
   Result: ODSG
 1 Function computeODSG (originEntries)
 2     instToOrigins := map<Context, map<Instruction, set<Origin> > >;
 3     memToOrigins := map<MemObj, set<Permission> >;
 4     forall origin o in originEntries do
 5         forall instruction i reached in o with context c do
 6             instToOrigins[c][i].add(o);
 7         end
 8     end
 9     forall each context c do
10         forall inst i in instToOrigins[c].keys() do
11             if i is read then
12                 p = READ;
13             else
14                 p = WRITE;
15             end
16             pts = getPts(i.operand(), c);
17             forall memory object m in pts do
18                 forall origin o in instToOrigins[c][i] do
19                     memToOrigins[m].add({o, p});
20                 end
21             end
22         end
23     end
24     postProcessing(memToOrigins);
25     handleSharedMemory(memToOrigins);
26 end
```

Origin-sensitive pointer analysis. We use a pointer analysis that is origin-sensitive and field-sensitive (developed by Liu et al. [83]). After identifying origin entries in the program, we set each origin entry as a context (`main` function is assigned the default context). Origin-sensitivity can help distinguish memory objects allocated or accessed in dynamic origins and the analysis is still scalable. Field-sensitivity can help resolve more accurate call targets and hence more precise points-to set.

Post-processing. Since the pointer analysis is field-sensitive, it is possible that the fields of a root memory object are mapped into different origins, depending on how they are accessed by instructions. However, memory objects that belong to different origins will be allocated to different heap regions (see Section 5.2.5 for more detail). And there is only one allocation call for the root memory object in the program. To resolve the conflicts, we need to merge the object fields to its root object. The result origin set of the root object is the union of origins of all its fields.

We also perform an escape analysis for memory objects that are only accessed by dynamic origins. Since each time the program execution enters a dynamic origin it is assigned a different origin ID, the dependent memory object from previous runs will cause false positives. This problem is addressed by escape analysis. We recursively check if the pointer of a memory object in dynamic origin escapes to heap memory objects in other origins or stack locations. Such memory objects are considered shared by all dynamic origins and we move them to a static origin.

5.2.3 Handling Shared Heap Objects

For memory objects that can be accessed by different origins, we will move them into new origins. As shown in Algorithm 3, we maintain a mapping between a permissions set to a new origin ID. All memory objects with the same permissions set will be moved into the same new origin. For the new origins, we do not keep the read/write permission since the old set has the information and the mapping preserves it. For example, `config` in Figure 5.1 is shared by the default origin *1* and the dynamic origin *0*. After this step, it will be moved to a new origin, say *2*.

The runtime checks need to know if an object is shared and which origin can share it. We use a 2d table S to store the origin sharing information, in which each row is an old origin, each column

is a newly created origin and the value represents the permission. For example, $S[i][j] = 0x1$ indicates that origin i has read permission ($0x1$) to memory objects in origin j.

Algorithm 3: Move shared objects into new origins.

```
   Result: Unique memory to origin mapping
 1 Function handleSharedMemory (memToOrigins)
 2     memToOrigin := map<MemObj, Permission>;
 3     originsToNew := map<set<Permission>, OriginID>;
 4     forall memory object m in memToOrigins.keys() do
 5         permissions = memToOrigins[m];
 6         if permissions.size() == 1 then
 7             memToOrigin[m] = permissions[0];
 8         else
 9             if permissions not in originsToNew then
10                 originsToNew[permissions] = getNewId();
11             memToOrigin[m] = originsToNew[permissions];
12         end
13     end
14 end
```

5.2.4 Instrumentation

Based on the origin data sharing analysis result, we then instrument the program to insert necessary runtime checks. For read/write to a heap object, DFO checks if the accessed memory belongs to the same origin as the runtime origin. If it does not, it further checks if the actual origin of the accessed memory is shared with the runtime origin, with the correct permission. Note that this check is performed per-object instead of per-4-bytes, coalescing checks for memory operations such as `memcpy`.

5.2.4.1 Instrument origin entries.

As shown in Figure 5.2, we need to instrument each origin entry to maintain a *thread local* runtime origin ID, `__rt_origin_id`, which will be used by later checks and heap allocations.

For static origins, we just assign the corresponding origin ID to this variable. For dynamic origins, we maintain a global atomic counter, `__dyn_origin_id`, assign its value to `__rt_origin_id` and increase the counter at the exit of the origin entry function.

5.2.4.2 Instrument heap allocations.

We handle heap allocations in two different ways. First, for all heap allocation calls (e.g., `malloc`,`calloc`, etc.) that are analyzed, we replace them with the call to our customized heap allocator, which requires an extra origin ID argument. Recall that in the origin data sharing analysis, each memory is assigned with an origin ID. Therefore, we can pass the computed origin ID of the memory object being allocated to the allocation call. For dynamic origins, instead of passing *0*, we pass the runtime origin ID to the allocation call. Section 5.2.5 will discuss the details of the allocation.

For heap allocations calls in third party libraries, we cannot replace them. Instead, we intercept the calls and redirect them to our customized allocator, with the default origin ID *1*. This is to ensure that third party code will not break the checks.

Origin-sensitivity. One challenge coming from origin sensitivity is that an object may be mapped to different origins under different origin contexts. As a result, we need to pass in different origin IDs to the allocation call under different origin contexts, so that the statically computed ODSG can be enforced at runtime. To address it, we attach a constant array to the allocation call, which stores the correct origin IDs given different origin contexts. The runtime origin ID is used to query the correct origin ID for allocation. For example, suppose an object *m* can be allocated by both default origin 1 and another static origin 2, and the one allocated in origin 2 is shared, so we move it into a new origin 3. Then the constant array attached to the allocation site of *m* is *[0,1,3]*, indicating that *m* should be allocated into the heap of origin 1 when the runtime origin ID is 1, and be allocated into the heap of origin 3 when the runtime origin ID is 2. Runtime origin ID for dynamic origin is mapped back to *0* in the querying function, so that the constant array can be fixed and small.

5.2.4.3 Instrument checks for reads/writes.

For each read/write to heap memory, we insert a runtime check immediately before it. The check function requires three arguments: memory address, runtime origin ID and the permission (read or write). This function will perform a few checks:

1. Query the origin ID for the memory being accessed. Check if it equals to the passed-in origin ID or *1*. If not, go to next check.

2. Check if the queried origin ID equals to *0*, if it does, report an error. Otherwise, go to next check.

3. Check if the origin of the memory is shared with any other origins. If not, report an error, otherwise, go to next check.

4. Check if it is shared with the passed-in origin. If not, report an error.

The second check is interesting. Recall that we reserve number *0* for dynamic origins when computing the cross-origin data-flow. However, when instrumenting the origin entry and heap allocations, we replace it with the runtime origin ID read from `__dyn_origin_id`. This guarantees that no memory object at runtime will have an origin ID *0*. For memory address that are not in heap or uninitialized memory, the origin ID queried from it usually is exactly *0*, so that we can detect the error.

Note that we only insert a single check for each read/write, no matter how large the memory object being accessed is. And we do not check intra-origin data-flow. This requires a customized heap allocator that can allocate memory objects of different origins into different heap regions, which we present next.

5.2.5 Origin-based Heap Allocator

The design of our origin-based heap allocator is inspired by [115]. We handle the allocation of small and large objects differently (threshold is set to 2MB, as in [115]). For large objects,

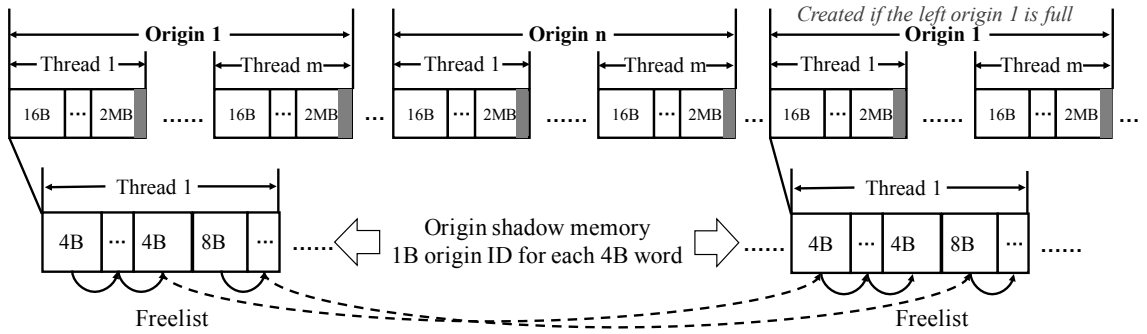


Figure 5.3: An overview of the origin-based heap allocator. The greyed rectangles represent guard pages at the end of sub-heaps.

we directly use `mmap` to allocate memory space, and store the allocation information (e.g., start address, size, etc.) and its origin ID in an ordered map. Given a memory location, we can determine which object it belongs to through comparing it with the lower bound and upper bound of each record in the map. Since the map is ordered, we can quickly locate the target object, and return the origin ID.

For small objects, we employ a more efficient design as illustrated in Fig. 5.3. The memory allocation, de-allocation, and origin ID querying of small objects are typically much more frequent than that for large objects. We first reserve a large memory space for each origin via `mmap`, called sub-heap. In each sub-heap, we classify the size of allocated objects into a set of classes, from 8B to 2MB. We put each heap object in the smallest bucket that can hold it. For example, an object of size 28 will reside in a bucket of size 32. For each class of size, we maintain a set of available buckets. When an object is allocated, if there is no available bucket in its origin's sub-heap, we will create a new sub-heap for that origin (e.g., Origin *1* at the right-hand side of Fig. 5.3).

We use a *freelist* to manage deallocations. When an object is deallocated, we add the memory location into the freelist of the corresponding class of size. Next time when an object with the same class of size is allocated, the freelist is checked to reuse memory space. The freelist is stored in the shadow memory, thus we do not need control blocks at the beginning of each heap object.

For large objects, we insert a guard page at the end of the object memory through the *mprotect*

system call. For small objects, we insert a guard page at the end of sub-heap. In this way, we only need to check the starting address of each accessed memory object, as overflow to another origin is no longer possible.

Querying origin ID. For large object, the origin ID can be obtained using its allocation information. For small object, the structure of the heap allocator implicitly stores the origin ID for each memory object. We use the below code to compute the origin ID of an accessed memory address of small object. `_originBegin` is the start address of the first sub-heap, `_bagShiftBits` is the number of bits for the size of each bag (the total memory for each class of an object size), `_numBagHeapShiftBits` is the number of bags in each sub-heap, and `_originMask` is simply `_numOrigin`-1.

```
1 offset = addr - _originBegin;
2 bagNum = offset >> _bagShiftBits;
3 heapIndex = bagNum >> _numBagHeapShiftBits;
4 originID = heapIndex & _originMask;
```

For frequent memory accesses, however, the above way of querying origin ID is still expensive. Therefore, we also use shadow memory to store the origin ID for each memory object. For each 4 bytes, we use 1 byte to store its origin ID (this limits the maximum number of origins to 256). For all heap allocations, this brings 25% memory overhead. Allowing larger maximum number of origins is at the expense of more memory overhead (e.g., we may use 2 bytes for origin ID to allow 65536 origins, with 50% memory overhead).

5.3 Implementation

We implemented DFO on LLVM 12. We generate a single bitcode file for each program. In the LLVM IR, top-level variables are represented in the SSA form, and address-taken variables are kept in memory and are not in SSA. All memory operations are executed through *Alloc*, *Load*, *Store* instructions, which are the sites for instrumentation. The intrinsic instructions such as `llvm.memcpy`, `llvm.memset` and `llvm.memmove` provide optimization for the corresponding library functions. As these functions access memory internally, we also need to instrument

them. This is done by instrumenting the call sites of these intrinsic instructions, without wrapper functions.

Analysis pass. DFO first extracts the origin entry annotations from the whole program IR. Then it performs an origin-sensitive pointer analysis with those origin entries. We use an 1-byte ID for each origin by default, but the implementation also allows 2-byte origin ID. We do not analyze 3rd party libraries used in the program, unless they are statically linked. We use the default origin ID *1* for all heap allocations in 3rd party libraries. We use *0* as an exception ID value since no valid origin has this ID at runtime.

Instrumentation of heap accesses. Our heap allocator takes the origin ID as an extra parameter to the memory allocation functions (*i.e.*, `malloc`, `realloc`, `calloc`). In the instrumentation pass, we replace these functions calls with our customized functions `xmalloc`, `xrealloc`, `xcalloc`, `xfree`, and pass the required memory size and the origin ID to them.

We only insert checks for heap accesses, by checking if the instruction operand can only point to heap objects.

Rust support. Since the backend representation of Rust is also LLVM IR, we implemented preliminary support for Rust to show that our approach can be generalized to other LLVM-based languages. Specifically, we model a set of Rust standard library APIs to adapt the pointer analysis, e.g., `rt::lang_start` and `alloc::alloc`.

DFO runtime and optimizations. Our heap allocator works by hooking the heap allocation functions we instrument (`xmalloc`, `xrealloc`, `xcalloc` and `xfree`). To handle heap allocations in 3rd party libraries, we also hook the default heap allocation functions (`malloc`, `realloc`, `calloc` and `free`) and pass them the default origin ID. The runtime library implements the functions of checking the origin IDs for heap objects, so that the instrumentation pass only needs to insert calls to these functions. To protect the shadow memory that stores the origin IDs, the checking functions will also ensure no writes to them outside of the origin-based allocator itself.

We implemented a set of optimizations to reduce the runtime overhead, without leveraging any

hardware features.

First, we reduce the number of branches as much as possible for runtime check functions. For example, we delay the check #2 described in Section 5.2.4.3 to when an error is detected. Since in normal program runs, the checking function will not execute the error reporting branch, delaying it can save a branch instruction. For checking origin sharing, though the 2d table is small, loading value from it is still expensive. However, each time the program execution enters a new origin, the row of the 2d table is fixed. For all instructions in the same origin, we only need to find the target permission from a small 1d array. In addition, for check #3, we originally have a bitmap to check if the target origin is shared. We found in initial experiments that this additional memory load adds more overhead than the checks (querying the 2d table) it saves. Therefore, we remove it and setting the entire row $S[i]$ in the 2d table to 0 to indicate that origin i is not shared. These optimizations reduce the runtime overhead by 50%.

Second, for loops, LLVM already performs a set of optimizations, one of which is moving the loads of all loop invariants outside of the loops. However, it is uncertain of the side-effect of our runtime checking functions, hence it chooses to not move them. To optimize it, we run an additional pass after instrumentation to also move the instructions of checking functions for loads of those loop invariants outside of loops correspondingly. This further reduces the runtime overhead on benchmarks with intensive loops by 20%.

5.4 Evaluation

In this section, we evaluate the efficiency and effectiveness of DFO. Our experimental environment has an i7 6700K 4.0GHz CPU and 48GB RAM running Ubuntu 16.04.

5.4.1 Benchmark

We collected a set of recent CVEs containing eight data-only vulnerabilities, as shown in Table 5.1. The Heartbleed CVE is also included. The statistics of these benchmarks are reported in Table 5.2, including the number of origin annotations.

We evaluated the runtime performance of DFO and compared with SoftBound and Address

Table 5.1: Benchmarks.

Program	CVE	Type	LOC	Description
OpenSSL	CVE-2014-0160	Overread	403.6K	Open source library that implements SSL and TLS protocols and various utilities (e.g., cryptography), included in all mainstream operating systems.
Mongoose	CVE-2021-26529	Overwrite	58.1K	Embedded web server and embedded networking library, widely used by hundreds of businesses.
ngiflib	CVE-2021-36531	Overread	1.5K	GIF picture format decoding library.
GNU cpio	CVE-2021-38185	Overwrite	72.5K	A general archiver utility in Unix-like systems for copying files into or out of a cpio sor tar archive.
libmysofa	CVE-2020-36150	Overread	6.8K	Reader for SOFA files, which is a spatially oriented format for acoustics.
	CVE-2020-36151	Overwrite		
libsixel	CVE-2019-20094	Overwrite	26.8K	A popular encoder/decoder implementation for SIXEL (a bitmap graphics format supported by terminals and printers), which also provides some converter programs.
reorder	RUSTSEC-2021-0050	Overwrite	0.3K	A utility crate for reordering a slice without an auxiliary array.

Sanitizer (ASAN) on the same benchmarks, using publicly available workloads. We acknowledge that both SoftBound and ASAN provide stronger security guarantee than DFO (and ASAN is usually used as a testing tool).

5.4.2 Analysis Results

In the rest of this section, we will discuss the experiment on each CVE separately.

OpenSSL. Heartbleed (CVE-2014-0160) is reported in OpenSSL. The root cause code is explained in Section 5.1, so we omit it here. In the function `do_server` called from `s_server_main`, there is a loop that keeps calling the callback function `sv_body`. Therefore, we set `sv_body` as a dynamic origin entry and set the rest as a default static origin. We run the instrumented binary

Table 5.2: Benchmark analysis statistics. The second column shows the number of manual annotations. The third column shows the number of origins created after DFO's analysis phase (*N* means dynamically created origins, determined by concrete inputs). The forth column lists the number of all instrumented heap accesses. The fifth column shows the ratio of shared heap accesses (note that different objects may be shared by different origins, here we only count if they are shared).

Program	Annotations	Created origins	Instrumented heap accesses	Ratio of shared heap accesses
OpenSSL	1	4+N	15,469	85.9%
Mongoose	1	2+N	253	77.9%
ngiflib	1	4+N	205	100%
GNU cpio	2	9+N	877	39.6%
libmysofa	1	4	991	39.5%
libsixel	1	2+N	1,274	0.7%
reorder	1	2	20	0%

and send malicious requests to it. We observe that the attack obtained mostly zero bytes while the server keeps running. The reason why the server does not terminate is that the maximum length of overread in *OpenSSL* is 16384, which still is an intra-origin access, thus is not reported. DFO guarantees that the overread can only read data of the same origin.

Mongoose. Mongoose is a popular embedded web server and networking library for C/C++. The vulnerability CVE-2021-26529 is reported in the executable binary *http-restful-server*, so we study its source code first, part of which is shown in Figure 5.4. It has a central event loop that keeps calling `mg_mgr_poll`, in which it reads incoming messages and executes different callbacks. Prior to this loop, it initializes some configurations and sets up the callbacks. Since there is no credential information initialized there, we set `mg_mgr_poll` as a dynamic origin entry and the rest as default origin.

The root cause code is in function `mg_http_serve_file`, as shown in Figure 5.5. With flooding requests, heap allocation may fail and return *nullptr*, resulting in a near-null pointer overwrite. This is dangerous in case of limited memory (e.g., embedded environment, which is exactly what *Mongoose* targets) as near zero memory regions usually store device configurations (on a normal machine, it will trigger segmentation fault since zero page is protected by default).

```
1 struct mg_mgr mgr;
2 mg_mgr_init(&mgr); // Initialise event manager
3 // Create HTTP listener
4 mg_http_listen(&mgr, s_listen_on, fn, NULL);
5 // Infinite event loop
6 for (;;) mg_mgr_poll(&mgr, 1000);
7 mg_mgr_free(&mgr);
```

Figure 5.4: Event loop of *Mongoose http-restful-server*.

```
1 struct http_data *d =
2     (struct http_data *) calloc(1, sizeof(*d));
3 d->fp = fp;
```

Figure 5.5: Root cause of CVE-2021-26529.

We run the instrumented binary and use *curl* to send *GET* requests asking for a file. We simulate a limited memory scenario by returning *nullptr* for *calloc* after a few requests. Since the accessed near-null memory address does not belong to a valid heap region and shadow memory region, our runtime check successfully prevented it and reported an invalid access error.

ngiflib. A GIF picture format decoding library. The vulnerability CVE-2021-36531 is reported in the executable binary *gif2tag*. There is a do-loop in the main function of its source code, in which the only function that handles external input is `LoadGif`. We naturally annotate it as a dynamic origin entry.

The root cause code is shown in Figure 5.6. Through a malicious crafted input, `GetByte` will read out-of-bound data until it sees a byte of data in wild memory that has satisfies the condition in line 6.

We run the instrumented binary with the available POC input and see immediately an error reported by the runtime check, due to an unexpected target memory origin ID (the memory region being overread does not belong to any origin yet, thus has an exception ID 0).

GNU cpio. This is a general file archiver utility with its associated file format. In function

```
1  for(i=0; i<g->ncolors; i++) {
2    g->palette[i].r = GetByte(g);
3    g->palette[i].g = GetByte(g);
4  }
5
6  static u8 GetByte(struct ngiflib_gif * g) {
7    if(g->mode & NGIFLIB_MODE_FROM_MEM) {
8      return *(g->input.bytes++);
9    }
10 }
```

Figure 5.6: Root cause of the CVE-2021-36531.

`process_copy_in` called by the main function, it calls `read_pattern_file()` first to read the pattern file, and runs a while loop that keeps calling `copyin_file()`. Since both functions handle external inputs, we annotate the former as a static origin entry and the latter a dynamic origin entry.

The root cause code for CVE-2021-38185 is shown in Figure 5.7. Due to an integer overflow at line 3, function `ds_resize` may do nothing, while the call site assumes that a larger memory chunk is allocated, thus an out-of-bound write occurs.

With a carefully crafted input, there could be a very large buffer overwrite that can even be exploited for arbitrary code execution. In fact, there is an available exploit POC that can successfully enter bash CLI. We run our instrumented version with this POC, and observe that the attack is successfully prevented, when it is trying to overwrite across the guard page inserted in our heap allocator.

libmysofa. This is a reader for spatial acoustic data files. Apparently, good origin entry points for this program should be a function that handles the external input. By studying the main function of its source code, as shown in Figure 5.8, we identify two interesting sites. With further checking, `printJson` contains only a set of calls to `fprintf`. Therefore, we annotate `mysofa_open` as a static origin entry.

We collected two CVEs for this program. First, CVE-2020-36150 is a large buffer overread oc-

```
while (next_ch != eos && next_ch != EOF) {
  if (insize >= strsize - 1) {
    ds_resize(s, strsize * 2 + 2);
    strsize = s->ds_length;
  }
  s->ds_string[insize++] = next_ch;
  next_ch = getc (f);
}

void ds_resize (dynamic_string *string, int size) {
  if (size > string->ds_length) {
    string->ds_length = size;
    string->ds_string = (char *) xrealloc((char *) string->ds_string,
     ↪  size);
  }
}
```

Figure 5.7: Root cause code of CVE-2021-38185.

```
// ...
printJson(stdout, hrtf, sanitize);
if (check) {
  // ...
  hrtf2 = mysofa_open(filename, 48000, &filter_length, &err);
  // ...
}
```

Figure 5.8: Source code of *libmysof*.

curred in function `loundness`, as shown in Figure 5.9. A maliciously crafted input has abnormal values for `hrtf->N` and `hrtf->R`, resulting a controllable overread. We run the instrumented binary with the available POC input, and observe an error reported by the runtime checks. The POC input only triggers an 124KB overread, which is still intra-origin overread. However, the wild memory region being overread is not used yet and hence has an exception origin ID 0. Therefore, the runtime checks can still prevent it.

The second one, CVE-2020-36151, is a buffer overwrite, caused by an integer underflow, as shown in Figure 5.10. Given a malicious input, `st->filt_len` could be *0*, resulting in a very

```
factor = loudness(..., hrtf->N * hrtf->R);

float loudness(float *in, int size) {
  float res = 0;
  while (size > 0) {
    res += *in * *in;
    // ...
  }
  return res;
}
```

Figure 5.9: Root cause code of CVE-2020-36150.

large value in the loop condition, since the integer type is *uint32_t*. With carefully selected values for `nb_channels` and `filt_len`, the size of overwrite can be up to 4GB. This apparently will cause invalid cross-origin accesses and touch the guard pages we set. Running the instrumented binary confirmed it, as the runtime checks terminated the execution.

```
int speex_resampler_reset_mem(SpeexResamplerState *st) {
  spx_uint32_t i;
  // ...
  for (i = 0; i < st->nb_channels * (st->filt_len - 1); i++) {
    st->mem[i] = 0;
  }
  return RESAMPLER_ERR_SUCCESS;
}
```

Figure 5.10: Root cause code of CVE-2020-36151.

reorder. This is a Rust crate for reordering slices without any auxiliary array. Since it is a library, we choose to analyze the test program provided by the CVE reporter, and we annotate the test function as a new origin entry, whereas the rest as a default static origin.

The root cause code for RUSTSEC-2021-005 is shown in Figure 5.11. Rust iterator has a `size_hint` function that can return an approximated size of the actual data vector. However,

this size can be incorrect and should not be trusted. If the value of `bla.size_hint()` is larger than the length of the actual data vector, it will return uninitialized memory. If the value is smaller than the length of the actual data vector, it will write out-of-bound at line 5 instead.

We tested the instrumented binary for both cases. For the former case, since the uninitialized data belongs to the same origin, DFO prevents the potential data leak. For the latter case, we use unsafe index function to suppress the Rust default bound checking, and successfully observed the cross-origin error due to the out-of-bound writes.

```
1 pub fn swap_index(bla:impl ExactSizeIterator<Item=u32>)->Vec<u32>{
2   // arr is a slice of vec
3   // and has same length as bla.size_hint()
4   for (i, a) in bla.enumerate(){
5       arr[a as usize] = i as u32;
6   }
7   // ...
8 }
```

Figure 5.11: Root cause code of RUSTSEC-2021-0050.

5.4.3 Performance Analysis

For *ngiflib*, *libmysofa*, *libsixel* and *GNU cpio*, we collected publicly available datasets. For *Mongoose*, we send 1000 various requests to measure the responding time. For *reorder* crate, we use the benchmark functions provided by itself. The results are shown in Table 5.3.

For OpenSSL, we use the *speed* utility included in it, which measures the throughput in fixed time. Figure 5.12 shows the results.

In the server benchmarks (i.e., *OpenSSL* and *Mongoose*), the overhead of DFO is less than 3%. The bottleneck of *cpio* and *libsixel* is more on I/O, hence DFO also has less than 3% runtime overhead. The running time of *ngiflib* is short, while DFO has initialization overhead for the origins heap, hence the overhead is higher. *libmysofa* and *reorder* contains mostly heap accesses (e.g., the

Table 5.3: Normalized runtime overhead on the benchmarks (SoftBound failed to compile most programs).

Program	ASAN	SoftBound	DFO
Mongoose	20.1%	-	2.5%
ngiflib	52.8%	153.5%	13.3%
GNU cpio	5.2%	23.4%	2.5%
libmysofa	38.7%	-	14.2%
libsixel	-0.7%	-	-0.1%
reorder	115.9%	-	20.7%
Average	38.7%	-	8.9%

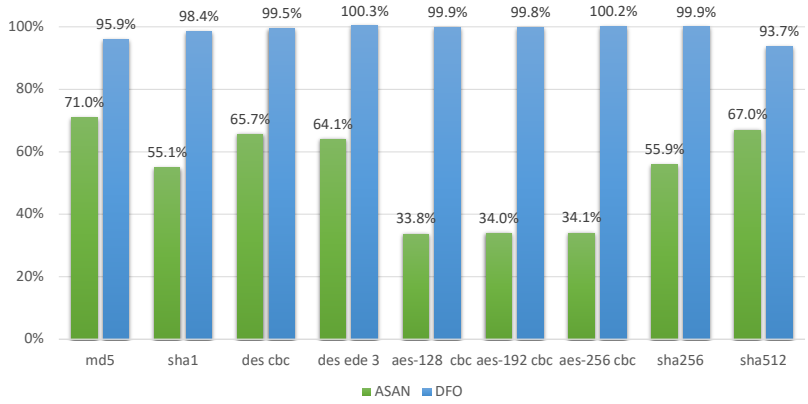


Figure 5.12: Throughput testing results on OpenSSL (the higher the better (native run is 100%).

benchmark functions in *reorder* simply allocate a huge array and manipulate it), therefore, we also observed higher overhead on them.

On the two benchmarks that SoftBound can successfully run, we observed approximately 10X higher overhead than DFO. While we build all benchmarks with ASAN in O2 mode (and enable all optimization flags that are usually disabled when using ASAN), we still observed very high overhead. On *OpenSSL*, it is 2X slower than native run and much slower than DFO.

With these results, we believe that DFO has the potential to be deployed in production. And future work can explore hardware features to further reduce the runtime overhead.

5.5 Limitations and Discussions

Our prototype tool has a few limitations. It does not secure external libraries without re-compiling (or statically linking) them. And though it uses state-of-the-art origin sensitive pointer analysis, the static analysis phase is still not perfectly precise, and may cause false origin sharing, leading to false negatives.

Limit of maximum number of origins. The current implementation only uses 1-byte for origin ID, thus at most 255 different origins (0 is reserved as exception value at runtime). We may also use 2-byte origin IDs, which will allow 65535 different origins, with 25% more memory overhead. In general, we will have only limited number of static origins. The number of dynamic origins may reach the limit, if an event loop keeps running. To address this, we currently recycle expired origin IDs, similar to process ID recycling in Linux kernel. And we consider that this can still significantly restrict the attack surface.

Intra-origin data-flow. The key assumption of this work is that an invalid data-flow is only meaningful if it touches sensitive data belonging to a different origin or the underlying system. Therefore, protecting intra-origin data-flow is not the focus. For complete data-flow protection, essentially we need to treat each memory object as a different origin (we need to distinguish memory allocations called by the same utility wrapper to avoid being too conservative).

Non-heap accesses. We only consider data-flow of heap objects in this work. Stack data accesses are always intra-origin and existing work can well protect it [75] (global data can be handled in a similar way). And according to recent statistics from Google [10], most severe memory corruptions are related to heap.

Performance tuning. There is still much room for optimizing the performance of DFO. For example, for heap object accesses with pointers of primitive types (e.g., *int**), DFO currently needs to check all of them. However, without pointer arithmetic and type casting on the base pointer, there cannot be out-of-bound read/write. An optimization on this may significantly reduce the runtime overhead. Another spot for optimization is loops. In practice, most runtime checks are placed inside loops (e.g., Figure 5.10). With scalar evolution analysis of the loop index, we may

skip inserting checks inside a loop that is free of out-of-bound errors.

5.6 Summary

We have presented DFO, a novel approach to prevent advanced non-control data attacks, by using an origin-sensitive static analysis and an origin-based heap allocator to detect illegal cross-origin data-flow efficiently at runtime. The evaluation on a list of recent CVEs shows that DFO can effectively secure real world applications with low runtime overhead. We believe that DFO is a promising security solution to data-only attacks, which are becoming an emerging threat.

6. CONCLUSION

In this dissertation, we presented three static analyses approaches that improves the state-of-the-art for software security at both compile-time and runtime:

- We presented a new approach for measuring code similarity, which combines static data-flow and control-flow with deep learning methods, significantly improving the accuracy of state-of-the-art. Meanwhile, driven by powerful GPUs, it outperforms traditional approaches in terms of time performance.

- We presented MIRAV, the first static analysis approach for detecting atomicity violations in Rust programs, which is a main source of Rust concurrency issues. It uses alias analysis to compute the set of atomic operations that could access the same variable from multiple threads, and builds static happens-before graph to check if they could execute concurrently. Then it applies heuristics to determine whether these atomic operations will cause atomicity violation. The evaluation on a benchmark of real world atomicity violation bugs shows that MIRAV is effective to detect them and has reasonable false positive rate.

- We proposed an effective and efficient defense against non-control data attacks by introducing the concept of origin. Specifically, we use origin-sensitive static analysis to compute the program data-flow, and instrument the program to check any invalid cross-origin data accesses. With our customized heap allocator, these checks incur very low runtime overhead ($< 9\%$ on average on our benchmark). The evaluation on a benchmark of recent CVEs shows that this approach effectively secures the target programs against all of the CVEs in the benchmark.

  With larger and larger CVE database, the first approach is important in searching vulnerable code in wild. Rust, as a memory-safe and race-free programming language, is adopted in many open-source projects and industrial software now. Our approach for detecting atomicity violations in Rust programs would help multi-threaded Rust programs achieve higher

level of safety. Meanwhile, since it is difficult to ensure bug-free for practical programs, approaches that ensure statically computed properties at runtime is necessary. Our last approach provides effective defense against real-world benchmark of CVEs, while incurring very low runtime overhead. These approaches together contribute to developing and building more secure modern software.

REFERENCES

[1] T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/Main_Page/, 2006. Accessed: 2016-09-02.

[2] TensorFlow: An open-source software library for Machine Intelligence. https://www.tensorflow.org/, 2016. Accessed: 2017-08-02.

[3] Google Code Jam. https://code.google.com/codejam/contests.html, 2016. Accessed: 2016-10-08.

[4] Deckard Github repo. https://github.com/skyhover/Deckard, 2017. Accessed: May/2017.

[5] The chromium projects: Control flow integrity. https://www.chromium.org/developers/testing/control-flow-integrity, 2018. Accessed: 2018-09-15.

[6] Heartbleed bug. http://heartbleed.com, 2018. Accessed: 2018-05-15.

[7] Chromium process models. https://software.intel.com/content/www/us/en/develop/articles/technical-look-control-flow-enforcement-technology.html, 2020. Accessed: 2020-08-31.

[8] Chromium process models. https://www.zdnet.com/article/microsofts-control-flow-guard-comes-to-rust-and-llvm-compilers, 2020. Accessed: 2020-08-31.

[9] Chromium cve statistics. https://www.cvedetails.com/product/15031/Google-Chrome.html?vendor_id=1224, 2021. Accessed: 2021-04-29.

[10] Project zero. https://googleprojectzero.blogspot.com/p/0day.html, 2021. Accessed: 2020-10-31.

[11] Firefox cve statistics. https://www.cvedetails.com/product/3264/Mozilla-Firefox.html?vendor_id=452, 2021. Accessed: 2021-04-29.

[12] Mir rfc. https://rust-lang.github.io/rfcs/1211-mir.html, 2022. Accessed: 2022-01-24.

[13] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[14] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[15] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[16] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 84–99, 2021.

[17] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[18] Brenda S Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.

[19] Brenda S Baker. Parameterized pattern matching: Algorithms and Applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

[20] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[21] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317. ACM, 2014.

[22] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings.,*

International Conference on, pages 368–377. IEEE, 1998.

[23] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, oseph M. JNash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, and Michael Franz. Hardware assisted randomization of data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, Lecture Notes in Computer Science, pages 337–358. Springer, 2018.

[24] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb):1137–1155, 2003.

[25] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2 (OOPSLA):1–28, 2018.

[26] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[27] Erik Buchanan, Ryan Roemer, Stefan Savage, and Hovav Shacham. Return-oriented programming: Exploitation without code injection. *Black Hat*, 8, 2008.

[28] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[29] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, pages 161–176, 2015.

[30] Scott A Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 193–204. ACM, 2017.

[31] S Carter, RJ Frank, and DSW Tansley. Clone detection in telecommunications software systems: A neural net approach. In *Proc. Int. Workshop on Application of Neural Networks to Telecommunications*, pages 273–287, 1993.

[32] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160. USENIX Association, 2006.

[33] K Mani Chandy, Jayadev Misra, and Laura M Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems (TOCS)*, 1(2):144–156, 1983.

[34] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[35] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186. ACM, 2014.

[36] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[37] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016.

[38] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[39] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *European Symposium on Research in Computer Security*, pages 37–54. Springer, 2012.

[40] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *NDSS*, volume 26, pages 27–40, 2012.

[41] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Stefan Wagner. Challenges of the dynamic detection of functionally similar code fragments. In *Software Maintenance*

and Reengineering (CSMR), 2012 16th European Conference on, pages 299–308. IEEE, 2012.

[42] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Detecting memory errors via static pointer analysis (preliminary experience). *ACM SIGPLAN Notices*, 33(7):27–34, 1998.

[43] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, 2003.

[44] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 393–407, 2010.

[45] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[46] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, volume 52, pages 58–79, 2016.

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

[48] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. *ACM Sigplan Notices*, 44(6):121–133, 2009.

[49] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. *ACM SIGPLAN Notices*, 38(5):338–349, 2003.

[50] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[51] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 321–330. IEEE, 2008.

[52] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[53] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, and K Words. Aries: Refactoring support environment based on code clone analysis. In *IASTED Conf. on Software Engineering and Applications*, pages 222–229, 2004.

[54] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[55] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[56] Reid Holmes and Gail C Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, pages 117–125. ACM, 2005.

[57] Richard C Holt. Some deadlock properties of computer systems. *ACM Computing Surveys (CSUR)*, 4(3):179–196, 1972.

[58] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[59] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 393–405. ACM, 2016.

[60] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *USENIX Security Symposium*, pages 177–192, 2015.

[61] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[62] Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–9. IEEE, 2010.

[63] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[64] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 55–64. ACM, 2007.

[65] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, pages 275–288, 2002.

[66] Elmar Juergens, Benjamin Hummel, Florian Deissenboeck, and Martin Feilkas. Static bug detection through analysis of inconsistent clones. *Software Engineering 2008*, 2008.

[67] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[68] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[69] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM*

International Conference on, pages 295–306. IEEE, 2015.

[70] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, pages 664–675. ACM, 2014.

[71] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017.

[72] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[73] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[74] Jens Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309. IEEE, 2001.

[75] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *OSDI*, volume 14, page 00000, 2014.

[76] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*, pages 712–717. Springer, 2012.

[77] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[78] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[79] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 249–260. IEEE, 2017.

[80] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *OSDI*, volume 4, pages 289–

302, 2004.

[81] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Mirchecker: Detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2183–2196, 2021.

[82] Bozhen Liu and Jeff Huang. D4: fast concurrency debugging with parallel differential analysis. *ACM SIGPLAN Notices*, 53(4):359–373, 2018.

[83] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. When threads meet events: efficient and precise static race detection with origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 725–739, 2021.

[84] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881. ACM, 2006.

[85] Peng Liu and Charles Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 299–309. IEEE, 2012.

[86] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. *ACM SIGOPS Operating Systems Review*, 40(5):37–48, 2006.

[87] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, 2006.

[88] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. Microstache: A lightweight execution context for in-process safe region isolation. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 359–379. Springer, 2018.

[89] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):

245–258, 2009.

[90] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 31–40, 2010.

[91] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3): 477–526, 2005.

[92] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[93] Robert O'callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, 2003.

[94] Gabriele Paolacci, Jesse Chandler, and Panagiotis G Ipeirotis. Running experiments on amazon mechanical turk. 2010.

[95] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 25–36, 2009.

[96] Nimrod Partush and Eran Yahav. Abstract semantic differencing via speculative correlation. *ACM SIGPLAN Notices*, 49(10):811–828, 2014.

[97] Karthik Pattabiraman, Vinod Grover, and Benjamin G Zorn. Samurai: protecting critical data in unsafe languages. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 219–232. ACM, 2008.

[98] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 763–779, 2020.

[99] Steven P Reiss. Semantics-based code search. In *Proceedings of the 31st International*

Conference on Software Engineering, pages 243–253. IEEE Computer Society, 2009.

[100] Chanchal K Roy and James R Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 172–181. IEEE, 2008.

[101] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[102] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[103] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[104] Hitesh Sajnani, Vaibhav Saini, and Cristina Lopes. A parallel and efficient approach to large scale clone detection. *Journal of Software: Evolution and Process*, 27(6):402–429, 2015.

[105] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168. ACM, 2016.

[106] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94, 2005.

[107] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[108] Gerhard Schellhorn, Oleg Travkin, and Heike Wehrheim. Towards a thread-local proof technique for starvation freedom. In *International Conference on Integrated Formal Methods*, pages 193–209. Springer, 2016.

[109] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):

699–742, 2014.

[110] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.

[111] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[112] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[113] Hossain Shahriar and Mohammad Zulkernine. Classification of static analysis-based buffer overflow detectors. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*, pages 94–101. IEEE, 2010.

[114] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Network. In *USENIX Security Symposium*, pages 611–626, 2015.

[115] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403. ACM, 2017.

[116] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[117] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011.

[118] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[119] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim,

Wenke Lee, and Yunheung Paek. HDFI: Hardware-assisted data-flow isolation. In *IEEE Symposium on Security and Privacy (SP)*, pages 1–17, 2016.

[120] Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):26, 2014.

[121] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 702–714. Association for Computing Machinery, 2016.

[122] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. Identifying functionally similar code in complex codebases. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[123] Jeffrey Svajlenko, Iman Keivanloo, and Chanchal K Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *Proceedings of the 7th International Workshop on Software Clones*, pages 16–22. IEEE Press, 2013.

[124] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *ICSME*, pages 476–480, 2014.

[125] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, pages 941–955, 2014.

[126] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, pages 1096–1103. ACM, 2008.

[127] Christoph Von Praun and Thomas R Gross. Object race detection. *Acm Sigplan Notices*, 36 (11):70–82, 2001.

[128] Stefan Wagner, Asim Abdulkhaleq, Ivan Bogicevic, Jan-Peter Ostberg, and Jasmin Ramadani. How are functionally similar code clones syntactically different? An empirical

study and a benchmark. *PeerJ Computer Science*, 2:e49, 2016.

[129] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[130] Hui-Hui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 3034–3040. AAAI Press, 2017.

[131] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[132] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[133] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337. IEEE, 2018.

[134] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Classification model for code clones based on machine learning. *Empirical Software Engineering*, 20(4):1095–1125, 2015.

[135] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[136] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving ap-

plication security with data flow assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, pages 291–304, 2009.

[137] Zeming Yu, Linhai Song, and Yiying Zhang. Fearless concurrency? understanding concurrent programming safety in real-world rust software. *arXiv preprint arXiv:1902.01906*, 2019.

[138] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[139] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V Chawla. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 793–803, 2019.

[140] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *USENIX Security Symposium*, pages 337–352, 2013.

[141] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208, 2008.

[142] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 121–132. IEEE, 2007.

[143] Minhaz F Zibran and Chanchal K Roy. Towards flexible code clone detection, management, and refactoring in IDE. In *Proceedings of the 5th International Workshop on Software Clones*, pages 75–76. ACM, 2011.