POLLUX: Safely Upgrading Dependent Application Libraries

Sukrit Kalra[*]
IIIT Delhi, India

Ayush Goel[*]
IIIT Delhi, India

Dhriti Khanna
IIIT Delhi, India

Mohan Dhawan
IBM Research, India

Subodh Sharma
IIT Delhi, India

Rahul Purandare
IIIT Delhi, India

ABSTRACT

Software evolution in third-party libraries across version upgrades can result in addition of new functionalities or change in existing APIs. As a result, there is a real danger of impairment of backward compatibility. Application developers, therefore, must keep constant vigil over library enhancements to ensure application consistency, i.e., application retains its semantic behavior across library upgrades. In this paper, we present the design and implementation of POLLUX, a framework to detect application-affecting changes across two versions of the same dependent non-adversarial library binary, and provide feedback on whether the application developer should link to the newer version or not. POLLUX leverages relevant application test cases to drive execution through both versions of the concerned library binary, records all concrete effects on the environment, and compares them to determine semantic similarity across the same API invocation for the two library versions. Our evaluation with 16 popular, open-source library binaries shows that POLLUX is accurate with no false positives and works across compiler optimizations.

CCS Concepts

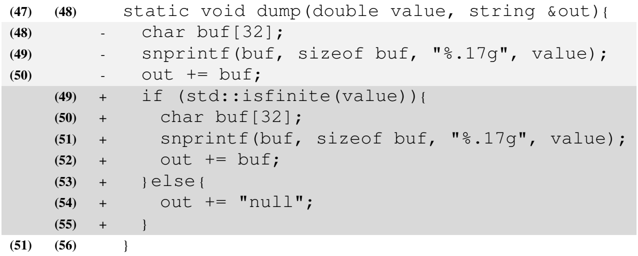
•**Software and its engineering → Software maintenance tools; Software testing and debugging; Dynamic analysis;** *Software evolution;* Software libraries and repositories;

Keywords

Software maintenance, Library upgrade, Dynamic binary analysis.

1. INTRODUCTION

In current times, library driven software development is a reality and use of third-party libraries is central to the development of a large number of applications. However, this software reuse comes at a cost—the included libraries can severely impact the maintainability of software systems. Evolution of third-party libraries may not always ensure backward compatibility, and may introduce new functionalities altering existing APIs across major upgrades. Thus, developers must keep constant vigil over library enhancements to ensure application consistency, i.e., the application retains its semantic behavior across library upgrades.



```
     static void dump(double value, string &out){
-      char buf[32];
-      snprintf(buf, sizeof buf, "%.17g", value);
-      out += buf;
+      if (std::isfinite(value)){
+        char buf[32];
+        snprintf(buf, sizeof buf, "%.17g", value);
+        out += buf;
+      }else{
+        out += "null";
+      }
     }
```

Figure 1: Dropbox's minor fix [6] could break applications.

A seven year study [44] of library release history in Maven Central, involving 150K binary JAR files, revealed that one third of all releases introduced at least one change that broke backward compatibility. This figure remained unaffected whether the library release was a major or a minor upgrade. Thus, choosing to update the library dependencies of an application is a double-edged sword, and demands thorough assessment of the effort needed to update the dependencies and the potential benefits achieved by updating.

For example, according to the JSON standard the values `NaN` and `Infinity` should be serialized to `null`. However, Dropbox's json11 library, which provides JSON parsing and serialization, used `snprintf` in its `dump` function to emit a string that was not compliant with the JSON standard. Thus, a minor fix as shown in Fig. 1, emitted significantly different JSON output for several applications, potentially breaking some functionality. This paper tackles the problem of whether a developer can safely upgrade a dependent library without affecting application functionality.

Prior work [30, 41, 43–45] has acknowledged the importance of dependencies in software management, and has empirically studied the effects of "update lag", "freshness", "quality", and "popularity" on dependency management. Teyton *et al.* [47] study library upgrades for JAVA software, but focus entirely on reasons and frequency of upgrades. However, none of the prior work focuses on the application developer's dilemma of whether library upgrades would break critical application functionalities. Also, most prior work rely on the analysis of source code, which may not even be available for several third-party libraries.

In this work, we present POLLUX, a framework that detects application-affecting changes across two versions of the same library binary, and provides feedback on whether the application developer should link to the newer version. POLLUX builds upon the observation that any critical, functionality-affecting API change in the newer version would manifest as a new or distinct concrete effect, such as memory writes and system calls. In absence of

[*] Both authors contributed equally.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950345

any such differences, the API invocation in the newer version is semantically similar to the older version. In other words, if the test cases elicit the same concrete effects upon invocation of the APIs across both versions, the execution is functionally similar; thus migrating to the newer version will not impact the application.

POLLUX leverages relevant application test cases (or the exhaustive library test suite, if it is open source) to drive execution through both versions of the concerned library binary, and records all concrete effects on the environment. POLLUX then uses a custom algorithm to compare these effects and determines the behavioral similarity across the same API invocation for the two library versions. In case of any dissimilarity, POLLUX lists the offending API call (in the newer version). While POLLUX is generic and applicable to all binaries, it is specific to libraries in the way that it assumes consistency of API interfaces across releases.

POLLUX is not a binary differencing tool; its goal is to detect semantic incompatibilities resulting from library upgrades. Note that POLLUX's effectiveness is contingent upon the exhaustiveness of the test suite that drives it. Test suite expansion will monotonically increase the differentials discovered by POLLUX. Further, any code path either added, removed or modified, if not traversed by the test suite, will result in POLLUX missing the changes. In general, determining behavioral differences across binaries is challenging for two major reasons. First, binaries might be obfuscated or compiled with separate optimization levels (say `-O0` and `-O3`) resulting in different binaries. Thus, graph isomorphism based techniques relying solely upon structural similarities in control flow also fail in light of these optimizations. Second, issue of scalability is ever present. Static and symbolic execution techniques perform precise semantic analysis, but suffer from issues of scaling to large binaries.

POLLUX overcomes these challenges using a path-sensitive dynamic binary analysis technique to identify behaviorally similar code segments in a library binary. Specifically, POLLUX generates a dynamic call graph for all library functions traversed during each library API invocation for both versions of the library. It then populates each node in the call graph with metadata to reflect concrete effects of execution associated with that function call, i.e., writes to stack and heap, and system calls observed during that specific function invocation. POLLUX aggregates these concrete effects into a function signature, i.e., a minimal set of concrete effects uniquely identifying the function invocation even across multiple executions for the same given inputs.

POLLUX leverages these function signatures to correlate similar functionality traversed during the API execution across the two library binaries. Any pair of non-matching function nodes indicates changes in either structural (i.e., code refactoring) and/or semantic functionality across the two binaries. POLLUX collates all such functionality changes for the given application library and notifies the application developer what specific API functionality is affected by linking to the newer library binary.

We have implemented a prototype of POLLUX for x86 binaries using Intel's PIN dynamic binary instrumentation framework [18, 39]. We have applied it to 16 popular, open-source libraries, and our evaluation shows that POLLUX correctly identifies semantic differences with no false positives for libraries under consideration. POLLUX's core signature extraction algorithm reports a precision of $> 99\%$ on call graphs generated for library APIs corresponding to adjacent versions for all 16 open-source libraries.

This paper makes the following contributions:

(**1**) We provide a practical design (§ 4) for POLLUX along with its novel dynamic binary analysis framework, which is robust, accurate, and precise.

```
(1)   int main(){
(3)     int *ptr;
(4)     ptr = (int*)malloc(sizeof(int));
(10)    *ptr = 25;
(11)    printf("%d", *ptr);
(12)    free(ptr);
(13)    return 0;
(14)  }
```

Figure 2: Example source code.

```
(1)  push   %rbp
(2)  mov    %rsp,%rbp
(3)  sub    $0x10,%rsp
(4)  mov    $0x4,%edi
(5)  callq  400510 <malloc@plt>
(6)  mov    %rax,-0x8(%rbp)
(7)  mov    -0x8(%rbp),%rax
(8)  movl   $0x19,(%rax)
(9)  mov    -0x8(%rbp),%rax
(10) mov    (%rax),%eax
(11) mov    %eax,%esi
(12) mov    $0x400709,%edi
(13) mov    $0x0,%eax
(14) callq  4004e0 <printf@plt>
(15) mov    -0x8(%rbp),%rax
(16) mov    %rax,%rdi
(17) callq  4004c0 <free@plt>
(18) mov    $0x0,%eax
(19) leaveq
(20) retq
(21) nopw   0x0(%rax,%rax,1)
```

(a) Example compiled w/ gcc `-O0`.

```
(1)  push    %rbx
(2)  mov     $0x4,%edi
(3)  callq   400520
            <malloc@plt>
(4)  mov     $0x19,%edx
(5)  mov     %rax,rbx
(6)  movl    $0x19,(%rax)
(7)  mov     $0x400719,%esi
(8)  mov     $0x1,%edi
(9)  xor     %eax,%eax
(10) callq   400530
            <__printf_chk@plt>
(11) mov     %rbx,%rdi
(12) callq   4004e0 <free@plt>
(13) xor     %eax,%eax
(14) pop     %rbx
(15) retq
```

(b) Example compiled w/ gcc `-O3`.

Figure 3: Example code depicting challenges in binary analysis.

(**2**) We implement POLLUX (§ 5) for x86 binaries and evaluate it (§ 6) on 16 open-source libraries to show its effectiveness in determining functional similarity across library versions.

2. MOTIVATION AND OVERVIEW

In this section, we motivate the need for POLLUX. First, we discuss two concrete scenarios describing the possible incompatibilities arising due to library upgrades. Second, we discuss the issues involving precise binary analysis that make the problem of detecting semantic differences challenging.

(**1**) **libxml crash**: libxml v2.7.6 encountered a segmentation fault with the upgraded zlib v1.2.3.5 because of a complete re-write of the gz* APIs, which read and write gzip files [13]. The fix to libxml is to check the version of zlib and to use the current code for `ZLIB_VERNUM` less than `0x1230`. However, since both libxml and zlib are widely deployed and numerous applications link to them dynamically, older versions of libxml and also other applications, without this fix, will fail with newer versions of zlib.

(**2**) **Winamp crash**: Winamp v5.666 build 3516 crashed due to a buggy component plugin (in_mp3.dll) [24]. The bug in the newer version of the plugin was an unintended consequence of fixing an older bug in the metadata editor. The proposed solution involved roll back of the plugin to build 3512.

The above examples highlight two issues commonly observed in software evolution. First, dependent API changes cascade all the way down to the application, which might crash if the appropriate changes are not handled gracefully. Second, feature enhancements in third party components, specially libraries and plugins, can easily introduce bugs leading to software crashes.

While newer technologies, like Docker [5], do remedy problems introduced by dependencies, they are, aimed primarily for software distribution alone. The core "dependency" issues (as discussed above) still remain unsolved in the context of system software, thereby motivating the need for POLLUX, which provides feedback to the developer on whether a dependency upgrade would affect or preserve the application's semantic behavior.

KEY CHALLENGES. Precise analysis of generic software binaries poses several major challenges [37].

- Binary formats do not strictly differentiate between code and data making their analysis difficult. Moreover, function boundaries are not well-marked because return instructions are not mandatory.
- Binaries lack rich data types available in the source, and may also lack symbolic information in release versions. Symbols and types can otherwise be used to improve precision of the analysis.
- Modern microprocessor instruction sets are large and complex, and many instructions have subtle differences, which, if ignored, can make an analysis unsound. In addition, presence of indirect jumps that calculate targets on-the-fly, and overlapping instructions that get resolved at runtime can make the analysis incredibly hard.
- The basic purpose of `call` and `ret` instructions is to execute a function call and a return, respectively. However, their usage to perform indirect jumps is abusive and can confuse the analysis.
- Lastly, some machine architectures allow self-modifying assembly code that can overwrite earlier code at the same address. Thus, the actual instructions executed may not be even present in the static disassembly of the binary.

OVERVIEW. Consider the example code shown in Fig. 2 where memory is allocated for an integer, updated to the value 25, and finally deallocated. The compiled output under optimizations `-O0` and `-O3` are shown in Fig. 3a and 3b, respectively. Despite significant syntactic differences among the two versions, writes to memory such as `movl $0x19, (%rax)` and system calls (`malloc`, `printf`, and `free`) are preserved along execution paths. POLLUX computes the signature of each function invocation in an execution of the program by capturing the set of memory writes and system calls performed by the function. The write set is a singleton set containing write of 25 to an address stored in `%rax` register. The set does not change across compiler optimizations, the optimizations can be thought of as idempotent operations with respect to the function signature set. POLLUX traverses through the call graph obtained from a program execution and performs the said activity repeatedly. At the end of the analysis, if no two function nodes are found to be behaviorally dissimilar, POLLUX declares the two versions to be behaviorally similar.

3. FORMAL OVERVIEW

A software binary can be considered to be a finite set of ordered pairs of inputs and execution traces (where trace is a sequence of events executed under the input; formal definition of a trace is presented in the ensuing text). Let this set be denoted by $\mathcal{B}$. For the purposes of this paper, the regression test suite or applications invoking libraries-under-test (LUT) define the domain of the input set, denoted by $\mathcal{I}$. Thus, for each input, $i \in \mathcal{I}$, we denote a set of execution traces of the program to be $\mathcal{B}_i := \{\tau | \langle i, \tau \rangle \in \mathcal{B}\}$. Such a definition subsumes alterations to the program via semantically equivalent code refactorings, compiler optimizations, speculative out-of-order execution semantics of the hardware and (in the case of concurrency) runtime scheduling. The set of all traces is denoted by $\mathcal{T}$. For each execution trace, an output value is produced. Since $\mathcal{I}$ and $\mathcal{T}$ are finite sets and assuming programs to be deterministic, the set of output values is also finite.

Given two binaries $\mathcal{B}$, $\mathcal{B}'$ of a software and a fixed regression suite $\mathcal{I}$, we wish to discover whether for each $b := \langle i, \tau \rangle \in \mathcal{B}$ (where $i \in \mathcal{I}, \tau \in \mathcal{T}$) there exists a $b' := \langle i, \tau' \rangle \in \mathcal{B}'$ (where $i \in \mathcal{I}, \tau' \in \mathcal{T}'$), such that b and b' have the same output and same state of system memory. This is commonly referred as *input-output equivalence* in the literature. There are a few scenarios relating to the above discussion that are relevant to the context of our problem *i.e.*, differential analysis of libraries:

(**1**) output of b and b' is same for a given i and $\tau = \tau'$, then clearly semantics have remain unchanged across the two versions,

(**2**) output of b and b' is same for a given i, however $\tau \neq \tau'$, then it mandates further analysis; while it is possible that the output of two traces is equivalent, the side-effects of traces may differ leading to different states of the system memory, altogether. Discovering precise side-effects is a hard problem in the average-case, while undecidable in the worst-case setting, and

(**3**) output of b and b' is not same for a given i; in such a case the application developer must be notified that it may not be entirely safe to upgrade the LUT.

3.1 Concrete Semantics

We begin by defining a simple low-level language that captures the essence of this work. The language uses pc as the program counter (note that we interpret pc to be pointing to the location of the instruction under execution), a finite set of integer registers $R = \{r_1, \cdots, r_n\}$, a store $m[.]$ that returns the contents at the memory location of the argument. The set of expressions in the language is denoted by $\mathbf{Exp}$. For simplicity, we do not specify the expressions in this language, although the expressions are allowed to contain pc, R, and $m[.]$ The set of program statements is denoted by $\mathbf{Stmt}$. A statement $s \in \mathbf{Stmt}$ can be one of the following:

- a variable assignment, $r_i := e$ with $r_i \in R, e \in \mathbf{Exp}$,
- a memory access, $m[e_1] := e_2$ or $r_2 = m[e_1]$, $e_1, e_2 \in \mathbf{Exp}$,
- a guarded `jmp`, $jmp\ e_1, e_2$, where $e_1, e_2 \in \mathbf{Exp}$, which jumps the pc to the address evaluated from e_2 given the (guard) e_1 evaluates to zero,
- a procedure call, $p()$.

A state s of a program defined in the above language is given by a triple: $\langle \mathcal{M}, l, fr \rangle$ where $\mathcal{M} : \langle \rho, \zeta \rangle$ captures the state of system's memory. Function $\rho : R \rightarrow \mathbb{Z}$ provides valuations to the registers, $\zeta : \mathbb{N} \rightarrow \mathbb{Z}$ provides the contents of the memory addresses, $l \in \mathbb{N}$ is the current address at which the control is , and $fr = (x_i, \cdots, x_{max-1})$ is the sequence of addresses indicating the frame structure of the stack. x_{max} is the maximum address to which stack can grow and x_i is the least address on the stack (stack grows downwards). A state transitions to a new state upon the execution of a statement in the following manner:

$$T[\![r_i := e]\!](s) := s[\rho(r_i) \mapsto \rho([\![e]\!](s))][l \mapsto s(l) + 1]$$

$$T[\![r_i := m[e_1]]\!](s) := s[\rho(r_i) \mapsto \zeta([\![e_1]\!](s))][l \mapsto s(l) + 1]$$

$$T[\![m[e_1] := e_2]\!](s) := s[\zeta([\![e_1]\!](s)) \mapsto [\![e_2]\!](s)][l \mapsto s(l) + 1]$$

$$T[\![\text{jmp}\, e_1, e_2]\!](s) := \begin{cases} s[l \mapsto s_l + 1] & \text{if } [\![e_1]\!](s) \neq 0 \\ s[l \mapsto [\![e_2]\!](s)] & \text{otherwise} \end{cases}$$

We assume that $[\![e]\!](s)$ is a deterministic evaluation function of statement e in state s. $T[\![e]\!](s)$ is essentially a state transformer function that produces a resultant state when statement e is executed from state s. The map $\mapsto$ updates/adds specific entries within a state. Finally, $T[\![q()]\!](s)$ is modeled by register assignment statements modeling two important aspects of the function call: stack frame allocation and deallocation. The body of the procedure q is modeled by statements of the language. At the time of allocation, the stack is extended by the frame size of $q()$. Thus, for stack allocation:

$$T[\![r_{fp} = r_{fp} - c]\!](s) = s[\rho(x_i) \mapsto (\rho(x_i) - c)(s)] \\ [f \mapsto (x_i - c, x_i \cdots, x_{max-1})]$$

r_{fp} is the register reserved for storing current frame pointer. Similarly, after the execution of the body of the function, the current stack frame at state s would be deallocated with $fr := (x_i - c, x_i \cdots, x_{max-1})$:

$$T[\![r_{fp} = r_{fp} + c]\!](s) = s[\rho(x_i) \mapsto (\rho(x_i) + c)(s)] \\ [f \mapsto (x_i \cdots, x_{max-1})]$$

A trace τ of a program is a sequence of states $s_0, \cdots s_{n-1}$ with s_0 as the start state. We assume that there exists function that maps program behaviors to outputs, $\mathcal{O} : \mathcal{B} \rightarrow \mathbb{Z}$. We define the notion of behavioral equivalence of two traces by the following definition:

Def. 3.1. Two traces τ, τ' are *strictly-similar* when on input i the state of memory is equal at their final states s_n, s'_n, i.e.,$s_n(\mathcal{M}) = s'_n(\mathcal{M})$ and $\mathcal{O}(b) = \mathcal{O}(b')$ where $b = \langle i, \tau \rangle$ and $b' = \langle i, \tau' \rangle$.

Although observing the output of a trace is self-evident, observing the effects of a library code execution on system's memory is not straightforward. For instance, programs could legitimately be writing addresses of memory locations as values into registers or memory locations. Such memory-address writes are bound to change even when the same program is executed multiple times. On account of such complexity, we define the notion of α*-similarity* of behaviors (as opposed to strict equivalence as defined above). The motivation behind α*-similarity* is to accommodate such address-based writes to registers/locations. Let $\mathcal{W}_\tau := \{\zeta[\![e_1]\!](s) | s', s \in \tau, T[\![m[e_1] := e_2]\!](s') = s\}$ be the multiset of values written into memory in the trace τ. Further, let $\mathcal{W}_{\tau|q}$ be the projection of writes performed by the procedure q called in τ. Register writes are not taken into account since we assume that often local or temporary values are written to registers. Even if that were not the case, scalability of POLLUX's analysis demands that we drop tracking writes to registers (per § 4).

Def. 3.2. Two behaviors b, b' from binaries B, B', respectively, on a given input i are α*-similar* when the following conditions hold: $\mathcal{O}(b) = \mathcal{O}(b')$ and $\frac{|\mathcal{W}_\tau \cap \mathcal{W}_{\tau'}|}{|\mathcal{W}_\tau \cup \mathcal{W}_{\tau'}|} = \alpha$.

We denote α-similarity of behaviors by a relational operator $\simeq_\alpha$, if b, b' are α-similar then $b \simeq_\alpha b'$. For a given input i, we abuse the notation and apply it for traces $\tau \simeq_\alpha \tau'$ when behaviors are found to be α-similar. Note that Def. 3.2 reduces to Def. 3.1 when $\alpha = 1$.

4. POLLUX

KEY IDEA. POLLUX relies on a key observation that any critical or functionality affecting change in third-party code is accompanied by corresponding side-effects, such as additional memory writes or system calls. In other words, key semantic behavior, such as memory writes external to the stack frame and system call sequence, remains unchanged despite compiler optimizations.

POLLUX takes as input the two library binaries and a test suite to drive execution of those binaries. For every test case, POLLUX generates a call graph with additional metadata, characterizing the signature for each function invocation. Next, POLLUX uses a custom algorithm that analyzes the two execution traces to identify semantically similar execution fragments. Execution segments that do not match are reported to the developer.

4.1 Execution Driver

The execution driver executes the test suite for both versions of the binaries. Specifically, it invokes the trace collector to start recording the effects of execution of each test case in the test suite. Once execution with the first binary is complete, the execution driver (i) serializes the recordings to external storage, (ii) dynamically links the test cases to second binary and executes them, and (iii) signals the trace collector to start recording again.

4.2 Trace Collector

Out of the several possible function level features, such as count of memory reads and writes, system calls, branching instructions, and indirect jumps, only critical, functionality-preserving memory writes and sequence of system call invocations remain unchanged in face of different compiler optimizations (`-O0` against `-O3`). This unambiguity is because both these features abstract out all the syntactic sugar (or operational mechanics), and are tightly linked to the functionality (or semantics) itself. Hence, POLLUX uses these two features to determine semantic similarity.

The trace collector is responsible for detecting and recording these two effects upon each test case execution. While it may be best to passively monitor these side-effects, it is not possible to do so for all effects, like writes to the memory. Recording such effects would entail instrumenting the entire execution environment, which would be prohibitively expensive. Thus, POLLUX leverages dynamic binary instrumentation for capturing effects of interest at a fine-grained level. This instrumentation preserves the intended execution effects of the binary, while also executing the hooks to capture additional metadata.

While binary instrumentation frameworks provide both coarse- and fine-grained hooks, POLLUX instruments the given binary at a per-instruction level granularity, thereby sacrificing low execution overhead in favor of high accuracy. For each instruction, POLLUX records the x86 instruction, and the corresponding data/address values. Thus, the instrumented binary upon execution enables POLLUX to keep precise track of: (1) data/addresses written to memory, with distinction between stack and heap, (2) system calls invoked along with their arguments, (3) calls to other imported library functions via the Procedure Linkage Table (`plt`), and (4) function return values, if available. At the end of test suite execution, the trace collector serializes the recorded values and trace analyzer is invoked, which is described next.

4.3 Trace Analyzer

The trace analyzer takes as input the serialized recordings for test suites corresponding to both library binaries, and determines semantic similarity using a layered two phase analysis. First, POLLUX deserializes each trace and creates a call graph with each node decorated with function-level metadata corresponding to memory writes and system call invocations. POLLUX then computes a precise signature for each function using metadata on memory writes and *asynchronous* system call invocations, and matches these function call nodes across the two execution traces for structural similarities based on (i) caller-callee relationship, and (ii) potential code refactoring. Second, if no dissimilar call nodes are observed, then POLLUX determines the execution sequences to be semantically similar iff the sequences of *synchronous* system call invocations observed across both the executions are same.

CALL GRAPH CONSTRUCTION. POLLUX determines call context per instruction, and groups instructions with the same context to build nodes in the call graph. Thus, while distinct invocations of the same function generate distinct nodes in the call graph, a recursive invocation generates a single node in the graph.

```
GRAPH_MATCH(a, b, MatchSet)
Input: a,b : Call nodes in execution graph of current library.
        MatchSet : Set of all matching node pairs.
Output: MatchSet : Set of all matching node pairs.

MatchSet = MatchSet $\cup$ {$\langle$ a, b $\rangle$};
$\Gamma_a = \Gamma_a - \Gamma_b$;   $\Gamma_b = \Gamma_b - \Gamma_a$;
$C_a$ = CHILDREN(a);   $C_b$ = CHILDREN(b);
foreach $x \in C_a, y \in C_b$: node_match(x,y) $\wedge \langle x, y \rangle \notin$ MatchSet do
    MatchSet = GRAPH_MATCH(x, y, MatchSet);
end
foreach $x \in C_a$ : node_match(x, b) $\wedge \langle x, b \rangle \notin$ MatchSet do
    MatchSet = GRAPH_MATCH(x, b, MatchSet);
end
foreach $y \in C_b$ : node_match(a, y) $\wedge \langle a, y \rangle \notin$ MatchSet do
    MatchSet = GRAPH_MATCH(a, y, MatchSet);
end
foreach $x \in C_a, y \in C_b$: $\langle$ x, - $\rangle \notin$ MatchSet $\wedge \langle$ -, y $\rangle \notin$ MatchSet do
    $\langle$ p, q $\rangle$ = FIND(x, y);
    MatchSet = GRAPH_MATCH(p, q, MatchSet);
end

FIND(a, b)
Input: a,b : Call nodes in execution graph for current library.
Output: $O$ : Set of matching node pairs from the two graphs
Initialize: $O = \emptyset$.

if node_match$^*$(a, b) then   $O = O \cup \{\langle$ a, b $\rangle\}$;
foreach $x \in \{a\} \cup$ CHILDREN(a), $y \in \{b\} \cup$ CHILDREN(b) do
    if $\langle x, y \rangle \neq \langle a, b \rangle$ then   $O = O \cup$ FIND(x, y);
end

GRAPH_SIMILARITY(r, r$'$, MatchSet)
Input: r,r' : Root node in library L and L', respectively.
        MatchSet : Set of all matching node pairs.
Output: Res: Boolean variable for match or no match
Initialize: MatchSet = $\emptyset$, Res = false.
foreach $\langle a, b \rangle \in$ FIND(r, r$'$) do
    MatchSet = MatchSet $\cup$ GRAPH_MATCH(a, b, MatchSet);
end
if $|MatchSet| > t_v \wedge \Omega_r = \Omega_{r'}$ then Res = true;
```

Algorithm 1: Match nodes with code refactoring.

4.3.1 Function-level Similarity

POLLUX constructs precise call graphs from deserialized recordings using a shadow execution context, and updates it on every new `call` instruction encountered in the trace. Simultaneously, it populates the nodes in the call graph with function-level metadata, such as writes to memory and *asynchronous* system call invocations. POLLUX considers an asynchronous system call equivalent to a memory write due to its non-blocking nature. Additionally, it maintains graph-level metadata that includes the exact sequence of *synchronous* system calls. The key observation here is that asynchronous system calls can be reordered with other operations, but synchronous calls must occur in sequence. Thus, semantic similarity must ensure that the sequence of synchronous system calls is preserved across both the executions. Any out of order synchronous system calls, which are blocking in nature (unlike an asynchronous call), could potentially indicate a different behavior, and hence, is not semantically similar.

Algorithm 1 depicts the steps POLLUX uses to determine function-level similarity across two call graphs. POLLUX leverages node metadata to compute a signature for every function in the two call graphs. It then uses the Sørensen-Dice index [20] (s_v) and an empirically determined threshold (θ) to determine a partial match between two call nodes. POLLUX also considers code refactorings, such as node splitting and inlining, while evaluating a function-level match. Observe that Sørensen-Dice index is essentially an instance of $\alpha-$*similarity* as noted in Def. 3.2.

FUNCTION SIGNATURE. The signature of a function must (i) be unique, and (ii) encapsulate semantic functionality. It follows linearly from Def. 3.2 that function signature of procedures must factor in the writes performed by it. It may not always be possible to track writes (writes performed by system calls), hence we conservatively treat asynchronous system call invocations as writes. Thus, for procedure q, its function signature is:

$$\Gamma_q = \mathcal{W}_{\tau|q} \approx \langle \widehat{\mathcal{W}}_{\tau|q}, \mathcal{S}_q \rangle \qquad (1)$$

where $\widehat{\mathcal{W}}_{\tau|q}$ is the multiset of all the memory writes of q that are observable, and $\mathcal{S}_q$ is the multiset of system call invocations in q.

The function signature for a specific call site, however, contains values which are execution dependent, such as addresses generated due to dynamic memory allocation; this introduces significant problem in deterministically assigning function signatures that do not fluctuate across repeated executions of the program. This problem stems from the fact that at the instruction level, POLLUX cannot distinguish between concrete data values and memory addresses, and accumulates both of them in the same write set. Keeping just the concrete data value and removing the addresses from the set of all memory writes would eliminate significant randomness in the function signature. Also, note that a function must export values out of its scope to perform useful functionality, which means the writes to function local variables result in no critical semantic behavior. Thus, POLLUX leverages process maps to determine address ranges for the current stack frame, and discards all write values within this range. Subsequently, function return values via the stack and registers (refer § 3) are not included in the function signature. Prior work [34] also notes that return values do not contribute significantly to the function signature.

SIGNATURE MATCH. A desirable signature matching scheme must ensure (i) few or no false negatives, and (ii) low false positives. However, an "exact" signature match solely from (1) can potentially result in high false positives in case of compiler optimizations, since these transformations may change the set of concrete data values produced by the function. Thus, POLLUX leverages a similarity-based match between the write sets generated by two functions a and b, based on the Sørensen-Dice index: $s_v = \frac{2|\Gamma_a \cap \Gamma_b|}{|\Gamma_a| + |\Gamma_b|}$, where set operators $\cup, \cap, +$ are applied separately to $\widehat{\mathcal{W}}_{\tau|x}$ and $\mathcal{S}_q$, $x \in a, b$. We consider α-similarity of functions only when the index is greater than a certain threshold ($\theta \in \mathbb{R}_{>0}$). The caveat is that due to reliance on a threshold value, it is possible that the value is not sufficiently high leading to false negatives (i.e., functions that should match but did not) or sufficiently low, leading to false positives (i.e., functions that should not match but did).

POLLUX uses function names, if available, to further improve the precision of the matching scheme. We observe that in a non-stripped binary where the symbol names are present, overloaded functions have different names. Furthermore, in C++, a function name is a mangled version of its class hierarchy and its parameters, which entails that every function name in the binary is unique. Thus, in a non-adversarial setting where symbols may be present, POLLUX utilizes the function names in addition to Sørensen-Dice index to match functions across binary executions. Finally, it is common for library developers to refactor code, i.e., inline or outline functions, or split a function into several smaller units. Prior art [27, 34, 49] has used function names as a heuristic in specific cases and POLLUX can leverage any of these more sophisticated techniques as its precision isn't contingent upon function names.

Note that with such function splitting, matching with function names is futile. In addition, the refactored functions make Sørensen-Dice index ineffective, since the index is based on function similarity rather than inclusion relationship. In such cases, using Sørensen-Dice index may lead to several false negatives. In order to make signature-matching more meaningful in the context of inclusion relationship, we introduce a new index $t_v = \frac{|\Gamma_a \cap \Gamma_b|}{min(|\Gamma_a|, |\Gamma_b|)}$ for functions a and b; this index, incidentally, also captures the results with the same precision as the Sørensen-Dice index would have for cases where inclusion relationship was absent. Thus, finally the node matching function is defined as:

$$node\_match(a,b) = \begin{cases} true: & t_v \geq \theta \\ true: & \lambda_a = \lambda_b \wedge \theta' < t_v < \theta \\ false: & Otherwise \end{cases}$$

where λ_x returns the name of the function x. When $t_v < \theta$ by a small margin (i.e., $\theta - \theta' = 5\%$), then, function names are matched.

Refactoring suggests that our matching mechanism should be capable of performing partial signature matches and also matches with the remainder of the signatures after a partial match is performed. In order to deal with partial matching and to maximize structurally meaningful matching, POLLUX uses the notion of residual signatures [42]. Specifically, whenever POLLUX matches the signatures of two functions, it also updates their current signatures with residual signatures (to be used for further matching) as follows: $\Gamma_a = \Gamma_a - \Gamma_b$ and $\Gamma_b = \Gamma_b - \Gamma_a$.

(1) **Inlining/Outlining**: Library functions are often inlined to achieve better performance. A function that is inlined adds its memory writes and asynchronous system calls data to its caller's signature, i.e.,if function a has its callee function b inlined in a newer version, it will result in the following signature of the new function: $\Gamma_{a'} = \mathcal{W}_{\tau|a} \cup \mathcal{W}_{\tau|b}$. Similarly, a function that is outlined as b in a newer version will have an opposite effect on the signature of the counterpart of b's caller function a as follows: $\Gamma_{a'} = \mathcal{W}_{\tau|a} - \mathcal{W}_{\tau|b}$. The inclusion index t_v captures this inclusion relationship allowing nodes to be matched correctly.

(2) **Splitting/Combining**: To achieve stronger cohesion and better maintainability, developers split functions. Splitting functions have an impact on the signatures and matching which is similar to outlining. More formally, when a function f that is split/combined into/from n functions $f_1, \ldots, f_n$, the following relation holds: $\Gamma_f = \mathcal{W}_{\tau|f_1} \cup \ldots \cup \mathcal{W}_{\tau|f_n}$.

EMPIRICAL THRESHOLD (θ). POLLUX randomly selects test cases for APIs that remain unmodified across the two neighboring versions of the library (corroborated by the commits), and determines t_v iteratively till the number of unmatched function nodes across the call graphs (corresponding to the two library executions) is less than 1%. In other words, at least 99% function nodes must match at this t_v. This final value of t_v is the threshold θ. The above iterative approach has the benefit that for most common cases, θ reflects the lower bound of similarity between semantically similar functions. Any value of $t_v \leq \theta$ that causes the number of unmatched function nodes to increase above 1% indicates, with a high probability, that the functionality has indeed changed.

4.3.2 Graph-level Similarity

In order to demonstrate similarity of two call graphs, POLLUX additionally handles the case when there are blocking system calls issued. It uses function $\Omega_x = \langle s_i, \ldots, s_j \rangle$ and $s_i, \ldots, s_j$ is the sequence of synchronous system calls observed in the execution.

4.4 Diagnosis

In case of dissimilar nodes, POLLUX traverses the function signature to determine the cause as either an extraneous data/address value or system call (or their order). In each case, the application developer receives a feedback indicating the offending API invocation, along with the entire call sequence leading up to the function responsible for the unmatched data/address value or system call that caused the dissimilarity.

COMPARISON WITH PRIOR ART. POLLUX's signature for matching functions across two execution graphs is robust and effective (as will be shown later in § 6). Unlike prior art [33–35,42], POLLUX leverages a layered approach to determining semantic similarity, and uses only the most critical side-affecting features, i.e., memory writes and system calls, which remain constant even across various compiler optimizations. Like BLEX [34], POLLUX also leverages dynamic binary analysis, but uses far fewer features to create succinct signatures. Additionally, BLEX aims for instruction coverage and generates random inputs for differential analysis that is not path-directed, thereby exploring infeasible paths and leading to several false positives. Unlike BinDiff [33] and [35], which use graph isomorphism techniques, POLLUX does not rely upon structural similarity and function names alone. Thus, POLLUX's signature built using dynamic mechanism is robust even under various compiler optimizations. Unlike [42], where a function signature includes *all* values read or written and are humongous, POLLUX uses only concrete data values and system call sequence to generate crisp function signatures.

4.5 Compiler Optimizations

Compiler optimization levels, such as `-O3`, are extremely aggressive and typically generate an execution graph that is significantly different from the one generated at level `-O0`. In fact, optimization `-O3` is akin to code refactoring at the assembly level. However, no amount of optimization should alter the functionality critical memory writes and sequence of system call invocations. In the absence of any structural similarity, POLLUX discards the function level signature matching and instead compares the aggregate set of write, and sequence of system call invocations. POLLUX leverages the index t_v for comparison across optimization levels. Since level `-O3` discards several intermediate memory writes, $\Gamma_a \subseteq \Gamma_b$ for two binaries a and b compiled for the same source code with levels `-O3` and `-O0` respectively. In other words, $t_v \approx 1$ indicates semantic similarity between binaries a and b.

Note that POLLUX's target is primarily application developers who include benign, third-party libraries, and typically, developers do not change compiler optimizations frequently for production-level code. Thus, matching semantic similarity across optimization levels is not the common case for POLLUX.

5. IMPLEMENTATION

We implemented a prototype of POLLUX based on the design described in § 4. While the trace analyzer and collector were automated and required ~900 lines of C++ to implement, the execution driver was triggered manually. We leveraged the Intel PIN [18, 39] dynamic binary instrumentation framework (v2.14) because of its ease of use in instrumenting the library binaries, and recording execution side-effects. We wrote a minimal "pintool", which is code that the PIN framework injects dynamically at selected points during instruction sequence, to extract relevant execution metadata and build a call graph for the given execution.

(1) **Call graph construction**. In assembly, function transitions, i.e., invocations and returns, happen via the `call*`, `jmp*` and `ret` family of instructions. POLLUX maintains a shadow stack of call context by leveraging PIN APIs to extract the function name at each transition instruction. However, we observed that a few functions did not have an explicit `ret` instruction, leading to anomalous call graphs. POLLUX overcomes this challenge by discarding the use of instruction-level instrumentation and switching to instrumentation at the granularity of a TRACE [1]. Since, a TRACE is part of exactly

[1] A TRACE is a straight-line instruction sequence with exactly one entry point. It usually ends with an unconditional branch, such as a call, return or unconditional jump. However, a TRACE may include multiple exit points

one function invocation, POLLUX invokes PIN APIs at the start of each TRACE to determine the function name, which helps to reliably maintain the shadow stack of call contexts. Note that PIN cannot reliably instrument functions in the presence of tail calls or when return instructions cannot reliably be detected [19]. Thus we did not use PIN's function-level instrumentation.

(**2**) **Detecting writes to stack**: Data values written to the stack mostly correspond to non-critical function local operations. Hence, it is important to discard them, so that the function signature uniquely identifies only critical functionality. Thus, POLLUX determines the address range available to the current process for writing to the stack frame, and removes from its write set any value written to an address within this range. To do so, POLLUX determines the process id for the currently executing test case, and reads the corresponding process maps from the `/proc` file system to determine the permissible stack range.

POLLUX leverages several optimizations to speed up the overall analysis and improve precision.

• **TRACE-level instrumentation**: Instruction instrumentation incurs significant overheads (due to dynamic code injection before every instruction for call graph construction), and also induces anomalies in construction as discussed earlier. POLLUX's use of PIN's TRACE-level instrumentation not only improves accuracy but also reduces the number of instrumentation points, which speeds up analysis by an order of magnitude.

• **Pruning the call graph**: POLLUX prunes the call graph for faster analysis. Specifically, the call graph starts at the API entry point and continues till the execution hits any `glibc` method invocations, like those corresponding to memory allocation and management, system calls, etc. The key observation here is that `glibc` and other system libraries, like `ldlinux`, provide access to fairly low-level functionalities to several system components, which change much less frequently compared with application libraries. Furthermore, a change in system libraries often necessitates upgrading the entire system and its dependencies. Not traversing the call graph before the API entry point and after the `glibc` function invocations significantly reduces the size of the call graph to be analyzed for semantic similarity.

• **Improving signature precision**: As explained in § 4.3.1, POLLUX identifies function local writes to memory to remove noise from the function signature. To further improve the signature, POLLUX executes each test case twice (linked to the same library binary), and takes an intersection between the side-effects observed across the two executions. The intuition here is that functionality preserving side-effects, such as writes and system calls, would remain unaffected and be present in the intersection.

6. EVALUATION

In § 6.1, we evaluate POLLUX for accuracy of detecting semantically relevant changes across several macrobenchmarks consisting of user applications. In § 6.2, we determine the precision of POLLUX's signature matching algorithm. In § 6.3, we determine the effectiveness of the various optimizations described earlier in § 5. In § 6.4, we check POLLUX's robustness across compiler optimizations. Lastly, in § 6.5, we present our experiences with POLLUX and demonstrate its utility in diverse conditions.

EXPERIMENTAL SETUP. All experiments were performed atop a VM having 4 VCPUs at 2.50 GHz, provisioned with 8 GB of RAM, and running 64 bit Ubuntu v12.04 with Intel PIN v2.14 installed.

DATA SET. We chose 16 popular, open-source C/C++ libraries from GitHub repositories (see Table 1), and randomly selected commit versions along with their test suites. We then manually inspected source code and the release notes corresponding to these commit versions and corroborated each such change. While POLLUX's analysis is entirely automatic, this manual involvement to validate our results limits the number of libraries analyzed.

EMPIRICAL DETERMINATION OF THRESHOLDS. We empirically determined the thresholds θ and θ' for each library (as described earlier in § 4.3). We observed that at $\theta = 0.95$, the fraction of unmatched nodes was $< 1\%$ across all libraries in our data set. Higher value of θ means a stricter check and would increase the fraction of unmatched nodes, while a lower value of θ indicates a more relaxed check and would have fewer unmatched nodes. Note that for a different corpus of libraries, θ might vary. We further selected $\theta' = 0.95 * \theta$.

6.1 Accuracy

We determine POLLUX's accuracy when one or more dependent libraries for a user application have changed. We consider POLLUX's output as accurate if it correctly determines changes in library code based on unmatched function nodes ($< 1\%$) in the call graphs and system call order, which must be preserved for semantic similarity. We capture ground truth for the concerned scenarios using commits from the open-source corresponding repositories.

We observe that barring a few security updates where code may get removed, as in the Heartbleed bug [22], most bug patches and feature enhancements either add new code or alter existing library code syntactically. We leverage the library test suites since the existing application test suites may not cover the entire gamut of functionality and run the suite with the two different versions. If the fraction of unmatched nodes is greater than 1% at $\theta = 0.95$, or there was a change in the system call order or count, POLLUX concludes a semantic change in the existing library version.

Table 1 reports our results. We observe that POLLUX manages to capture even subtle changes, such as in json11, where a mere 4 line change in the `dump` function (see Fig. 1) introduced significant changes across several other API executions, leading to $\sim$7% unmatched nodes across the test suite.

POLLUX correctly detects semantic changes in 28 out of the 30 scenarios tested. POLLUX reports false alarms for some `Capstone` and `Valijson` test cases. On manual inspection of the commit logs, we observed that the `Capstone` library did not have a test case that traversed the modified code. Since POLLUX leverages dynamic analysis, paths not traversed in the code are not validated for semantic changes. Hence, POLLUX reported no change in semantic similarity across the two `Capstone` versions. In `Valijson`, we observed that the modified API introduced no extra nodes. Furthermore, the code introduced only a conditional statement, which was not traversed by any of the test cases, similar to the `Capstone:ppc` scenario. Thus, POLLUX correctly reports semantic similarity in each of the 30 cases, thereby having a 100% accuracy for libraries under consideration.

6.2 Precision

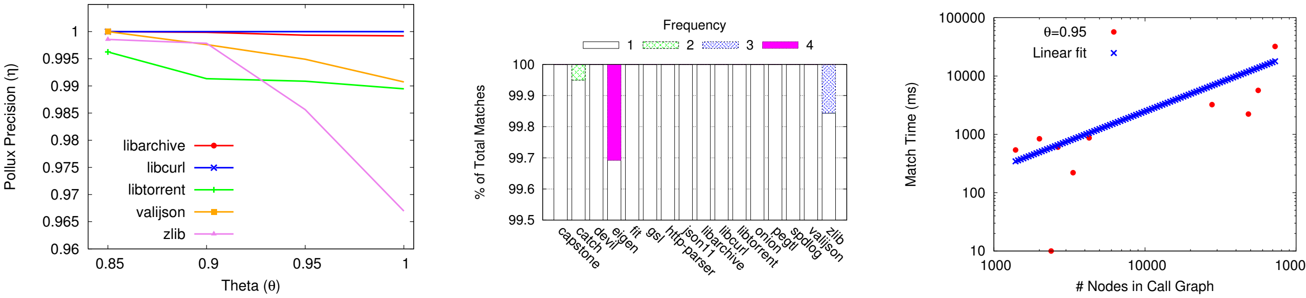
We determine the precision of POLLUX's function signature matching algorithm under the setting where there are no semantic differences for a given library API across the two versions. In such a scenario, we define precision as: $\eta = 1 - (n/N)$, where N is the total number of nodes in the call graph, and n is the unmatched nodes across the two versions of the library.

We select test cases for each library where the API does not change semantically across the two versions. We corroborate this

as long as they are conditional. If PIN detected a branch to a location within a TRACE, it will end the TRACE at that location and start a new TRACE.

Table 1: POLLUX accuracy across patches, minor and major revision changes for various libraries at $\theta = 0.95$. Note that POLLUX detects a semantic change if unmatched nodes are $>1\%$ and system call order is not preserved. $*$ indicates all APIs in the test suite.

#	Application	Library	API	Version / Commit		# Nodes			System calls	Detection
				Old	New	Total	Unmatched	%	preserved	
1	ArmExec	Capstone [1]	arm	3.0.3	3.0.4	5095	90	1.766	✗	✓
2	ArmExec	Capstone	mips	3.0.3	3.0.4	1291	20	1.549	✓	✓
3	ArmExec	Capstone	ppc	3.0.3	3.0.4	1856	6	0.323	✓	✗
4	ArmExec	Capstone	x86	3.0.3	3.0.4	4045	177	4.376	✓	✓
5	ArmExec	Capstone	xcore	3.0.3	3.0.4	1233	31	2.514	✗	✓
6	Visual Studio	Catch [2]	$*$	1.2.0 build 45	1.3.5 latest	26032	4432	17.025	✓	✓
7	Gazebo	DevIL [4]	$*$	1.7.8:$1f0d$	1.7.8:7241	64	4	6.250	✓	✓
8	TensorFlow	Eigen [7]	$*$	3.2.7	3.2.8	366	4	1.093	✓	✓
9	TensorFlow	Eigen	$*$	3.2.7	3.2.8	3340	60	1.796	✓	✓
10	TensorFlow	Eigen	$*$	3.2	3.2.8	251	81	32.271	✓	✓
11	Boost	Fit [8]	$*$	$e3bf390$	$52b54cd$	76	6	7.895	✓	✓
12	Boost	Fit	$*$	$52b54cd$	$66976be$	230	11	4.783	✓	✓
13	SageMath	GSL [9]	$*$	1.9	1.10	11932	354	2.967	✓	✓
14	SageMath	GSL	$*$	1.9	2.0	11932	787	6.596	✓	✓
15	NodeJS	http-parser [10]	parse_url	$ab0b16$	$7d75dd$	99	2	2.020	✓	✓
16	NodeJS	http-parser	execute	$ab0b16$	$7d75dd$	71786	7161	9.975	✓	✓
17	Dropbox	json11 [6]	dump	019364	$0e8c5b$	534	41	7.678	✓	✓
18	Dropbox	json11	parse	019364	$0e8c5b$	1077	74	6.871	✓	✓
19	CMake	libarchive [11]	$*$	3.0.2	3.1.0	14991	8327	55.55	✓	✓
20	AlsaPlayer	libcurl [3]	$*$	7.20.0	7.21.1	1372	250	18.222	✓	✓
21	AlsaPlayer	libcurl	$*$	7.20.0	7.47.0	1008	396	39.286	✓	✓
22	Deluge	libtorrent [12]	$*$	1.0	1.1	1459	554	37.971	✓	✓
23	Aisoy	Onion [14]	response_new	$c812b35$	$88a659$	786	48	6.107	✓	✓
24	Quinoa	PEGTL [16]	$*$	1.2.1	1.2.2	34010	3013	8.859	✓	✓
25	Quinoa	PEGTL	$*$	1.1.0	1.2.2	25182	1535	6.096	✓	✓
26	CBDM	spdlog [21]	$*$	$c7864ae$	$e248895$	1880	220	11.702	✓	✓
27	Puppet	Valijson [23]	$*$	$b241b37$	$e9b5016$	3343	17	0.509	✓	✗
28	OpenSSH	zlib [25]	$*$	$2689b$	$c58f7a$	1648	93	5.643	✓	✓
29	OpenSSH	zlib	$*$	1.2.7	1.2.8	1278	85	6.651	✓	✓
30	OpenSSH	zlib	$*$	1.2.5	1.2.8	1204	487	40.449	✓	✓



(a) Variation in precision with θ.

(b) Frequency of node matches at $\theta = 0.95$.

(c) Variation in graph comparison time.

Figure 4: Variation in POLLUX's precision and other properties.

Table 2: POLLUX Precision with 16 libraries at $\theta = 0.95$.

#	Library	Version / Commit		# Nodes		Precision
		Old	New	Total	Unmatched	
1	Capstone	$b560c2$	$c508c4$	56076	16	0.999
2	Catch	$ae5ee2c$	$f895e0d$	27753	0	1.000
3	DevIL	1.7.8:$cdc3$	1.7.8:$c806$	264	2	0.992
4	Eigen	$a1430dd$	$08e2e06$	2323	11	0.995
5	Fit	$953b721$	$ec7a043$	2002	0	1.000
6	GSL	$5c14$	$002f$	13851	0	1.000
7	http-parser	$bee48$	$4e382f$	72537	0	1.000
8	json11	$a6a66$	$e1d5b$	2395	0	1.000
9	libarchive	3.1.0:5818	3.1.2:$19f2$	15222	5	0.999
10	libcurl	$3c2e$	$e506$	2652	0	1.000
11	libtorrent	1.0.9	1.1.0	4270	39	0.990
12	Onion	$d7eb5b7$	$801bb9b$	4540	0	1.000
13	PEGTL	$02ba$	$9e3b$	44078	0	1.000
14	spdlog	$c0c5c01$	$a6a661e$	587	0	1.000
15	Valijson	$1ade1c5$	$b241b37$	3343	17	0.995
16	zlib	1.2.7.2	1.2.7.3	1281	2	0.998

API similarity by inspecting commits to the library repository. We run POLLUX for these test cases and measure the number of nodes that match across the API call graphs for the two executions. We observe that on average POLLUX reports a high precision (>0.99) across all test cases (see Table 2). We also note that `libtorrent` reports a high number of unmatched nodes, because the particular test case downloads a file, and thus several operational parameters, such as available network bandwidth and bytes downloaded, change significantly across two execution traces, thereby causing POLLUX to report the high number of unmatched nodes.

(1) Variation in precision with θ: We select five libraries from our data set and plot the variation in POLLUX's precision for different values of θ = 0.85, 0.90, 0.95, and 1.00. Fig. 4a plots the results. We observe that as θ increases, precision η decreases because in a stricter setting fewer function nodes match.

(2) Frequency of node matches: We measure the frequency of matches for nodes whose function signature matched to determine the aggressiveness of POLLUX's signature matching algorithm. Fig. 4b plots the results for all libraries in our data set at $\theta = 0.95$. The y-axis in the graph starts at 99.5%. We observe that across all libraries POLLUX correctly matches $>$ 99.5% of nodes, which

Table 3: Effect of pruning glibc nodes at $\theta = 0.95$.

#	Library	Version		# Nodes (w/o opt.)		# Nodes (w/ opt.)		Savings
		Old	New	Total	Unmatched	Total	Unmatched	(%)
1	GSL	$5c14$	$002f$	25966	0	13851	0	46.66
2	json11	$a6a66$	$e1d5bc$	7814	63	2395	0	69.35
3	Onion	$d7eb5$	$801bb$	725	8	699	0	3.59
4	PEGTL	$02ba$	$9e3b$	53621	257	32706	0	39.01
5	zlib	1.2.7.2	1.2.7.3	1969	3	1281	2	34.94

Table 4: Effect of re-execution on signature size $\theta = 0.95$.

#	Library	Version/ Commit	Avg. Signature Size		Savings
			w/o opt.	w/ opt.	(%)
1	Capstone	$b560c2$	21.61	21.11	2.34
2	Eigen	3.2.7	52.37	43.62	16.72
3	Fit	$9e132$	17.81	16.13	9.42
4	Onion	$801bb$	17.51	15.43	11.90
5	libtorrent	1.0.9	22.14	20.82	5.96

Table 5: Effect of re-execution on precision at $\theta = 0.95$.

#	Library	Version		# Total Nodes	Unmatched Nodes		Savings
		Old	New		w/o opt.	w/ opt.	(%)
1	Eigen	$a1430$	$08e2e$	2323	267	11	95.88
2	libarchive	5818	$19f2$	15222	1833	5	99.73
3	libtorrent	1.0.9	1.1.0	4270	1460	37	97.47
4	Onion	$d7eb5$	$801bb$	4540	686	0	100.00
5	PEGTL	$02ba$	$9e3b$	44078	2161	0	100.00

indicates the effectiveness of the algorithm. Only Catch, Eigen, libarchive and zlib had minuscule number of nodes with multiple matches due to sparse or common function signatures.

(**3**) **Variation in graph comparison time**: We determine variation in the match time with increase in nodes in the call graph. Fig. 4c plots the results for all libraries, except GSL and libarchive, at $\theta = 0.95$. We observe that in general as the nodes increase, the match time increases exponentially. However, both GSL and libarchive show much higher matching times than normal, which is possible since matching also depends on the structure of the graph. Code refactoring can also alter the graph structure.

6.3 Effectiveness of Optimizations

(**1**) **Pruning glibc nodes**: We selected five libraries and executed their entire test suites with and without this optimization enabled. Table 3 lists the results. We observe that pruning glibc nodes alone not only decreases the number of unmatched nodes, but also significantly reduces the graph size by an average of ∼39% across the five libraries under consideration.

(**2**) **Library re-execution**: We selected five libraries and executed their test suites twice with POLLUX. We then took the intersections of the signature values to determine the function fingerprints, and subsequently the improvement in precision with and without this optimization enabled. The rows in Tables 4 and 5 indicate that the intersection of function signatures from two executions provides significant savings, and reduces function signature by an average of 9.27% across the five libraries. Further, this optimization decreased the unmatched nodes by $> 95\%$ across the libraries.

6.4 Compiler Optimizations

We run POLLUX against five library test suites dynamically linked with corresponding libraries compiled with -O2 and -O3 compiler optimizations, and measure its effects on POLLUX's precision. Table 6 lists the results. We observe that even across the two optimization levels, POLLUX retains reasonable precision for most libraries except zlib. However, it drops significantly from $>$ 99% (per § 6.2) observed when determining semantic similarity for binaries with the same optimization level. Note that -O3 is an aggressive optimization level and includes function inlining, unswitching loops, among others. Thus, $\theta = 0.95$, which indicates an error margin of just 5% in similarity, is insufficient across optimization levels. We therefore need to recalibrate θ for detecting semantic similarity across optimization levels.

Table 6: Effect of optimization on precision at $\theta = 0.95$.

#	Library	Version/ Commit	# Nodes		Unmatched nodes (-O2)	Precision
			-O2	-O3		(%)
1	http-parser	$5651a$	72455	72465	1786	97.54
2	json11	$afcc8$	7803	7855	250	96.80
3	libcurl	7.47.0	2652	2508	1252	52.79
4	Onion	$51ceb$	699	600	140	79.97
5	zlib	50893	1701	1262	1065	37.39

Table 7: t_v across -O0 and -O3 compiler optimizations

#	Library	Version	Avg. Signature Size		Signature Intersection	t_v
			-O0	-O3		
1	Capstone	$d17fc$	399613	226594	191841	0.847
2	Eigen	3.2.7	2598	2142	1990	0.929
3	Fit	$9e132$	30	30	30	1.000
4	http-parser	$5651a$	758369	442938	428776	0.968
5	json11	$afcc8d$	78758	10900	10663	0.978
6	libtorrent	$508cc$	4270	4069	3889	0.956
7	Onion	$51ceb$	2778	2364	2256	0.954
8	spdlog	$c6f8f$	467	467	467	1.000
9	Valijson	$e9b50$	435678	325634	320004	0.983
10	zlib	50893	1033231	522940	522265	0.999

We now briefly compare POLLUX's effectiveness with BinDiff [33, 35] and BLEX [34]. Since, BinDiff is a proprietary tool, and BLEX's source and binary are unavailable, we use accuracy numbers available in [34]. Since POLLUX's signature matching is ineffective across huge structural changes, it leverages mechanism as described in § 4.5 to detect semantic similarity across optimization levels. Table 7 lists t_v observed for 10 libraries from our data set across -O0 and -O3 compiler optimization levels. We observe that POLLUX determines semantic similarity with 96.1% accuracy on average across the 10 libraries. In contrast, BLEX and BinDiff report an accuracy of ∼50% across the same optimization levels (per [34]), thereby making POLLUX's accuracy comparable to both BLEX and BinDiff.

6.5 Case Studies

(**1**) **json11**: json11 is an open-source C++ library from Dropbox. Commit $0e8c5ba$ fixes a bug where values like NaN and Infinity were serialized to non-compliant values by snprintf causing the deserializer to fail. The fix involved adding a condition which would return null if the number failed the std::isfinite check. Even this small fix resulted in an increase in the number of unmatched nodes, especially in test cases concerning numbers. A test case traversing the True branch of the conditional had 25 of 286 nodes (or 8.75%) left unmatched in the call graph. In contrast, the False branch had 11 of 235 nodes (or 4.68%) left unmatched. We observed that across the entire test suite, 1716 nodes were generated, of which 125 were unmatched (or 7.28%). Also, a change in the total number of nodes was observed only for test cases that executed the changed code path.

(**2**) **zlib**: zlib is a hugely popular library used by git, rsync, libpng, etc. Commit $c58f7ab$ replaced unsafe functions like strcpy with safer alternatives like snprintf. Such changes, wherein a function has been replaced results in significantly different memory writes. For example, the return values of the two functions (although unused), are totally different. strcpy returns the destination character array, while snprintf returns the number of bytes written. Also, unlike strcpy, snprintf appends a NULL byte to the buffer. POLLUX detects this change and reports 93 out of 1648 nodes (or 5.64%) as unmatched in the graph.

(**3**) **http-parser**: HTTP-Parser is a dependent library for NodeJS. Commit $4e382f9$ had only minor changes to the documentation and did not modify the source code. POLLUX generated a total of 72537 nodes for the entire test suite that match perfectly at $\theta = 0.95$, thereby indicating semantic similarity.

Commit $7d75dd73$ introduced support for Zone ID in IPv6 scoped addresses in the `http_parse_host` API. Only one test case (`test_parse_url`) in the entire test suite invoked this API. This test case generated a total of 99 nodes of which 2 were unmatched (2.02%) causing POLLUX to flag the change. For all other test cases, a total of 72433 nodes were generated that matched perfectly at $\theta = 0.95$, thereby pin-pointing the modified API.

Commit $0097de$ changed the way the tokens are parsed by the library's `http_parser_execute` API. Across the test suite, 3 tests did not invoke this API and generated a total of 154 nodes that matched completely. Further, 6 tests generated a total of 2337 nodes, 10 of which were unmatched (0.42%) causing POLLUX to conclude that they did not traverse the affected code path. Lastly, 2 tests that traversed the changed code path, generated 6355 nodes, of which 6300 were unmatched (99.13%) indicating changes to the API. Further, decreasing θ to 0.9 caused all nodes to match.

(**4**) **Capstone**: Capstone is a disassembly framework popular in the reverse engineering community. Commit $c508c4a0$ added support for the Travis CI build system and did not affect the source code. POLLUX reported a near perfect match across the entire test suite. Of the 56076 nodes generated, only 16 were unmatched (0.03%), indicating a trivial change.

Across a minor release upgrade from v3.0.3 to v3.0.4,seven cases in the test suite had only 0.76% unmatched nodes indicating insignificant changes to these modules. The remaining 5 test cases for the ARM, MIPS, x86 and XCore modules generated 15350 nodes, of which 385 were unmatched (2.51%). All changes except those in the PowerPC module were correctly detected by POLLUX. Upon further inspection, we observed that additional error checking conditions were introduced. However, the test cases did not traverse this newly introduced code change and thus, POLLUX reported a false positive for PowerPC.

7. LIMITATIONS AND FUTURE WORK

- POLLUX cannot handle multi-threaded interleaved executions. This limitation stems from its design, which requires deterministic comparison of side-effects resulting from individual test case executions. In contrast, multi-threaded executions introduce significant non-determinism in the set of captured side-effects.

- Since POLLUX leverages test suites for its dynamic binary analysis, it cannot detect changes in code paths not traversed by the test case. For example, POLLUX cannot detect bug patches that comment out an entire code path, such as the OpenSSH bug [15], where the vulnerable code in the client was completely disabled.

- Like prior work [33–35, 42, 50], POLLUX cannot reliably detect semantic similarities where side-effects involve random numbers, time of day, etc. However, in our observation, most critical features in mature libraries do not involve significant randomness.

- POLLUX does not include writes to registers or stack in its function signatures. Thus, it may miss values passed via stack or when entire function computation leverages registers alone.

- There can be several programmatic ways to encode the desired functionality. For example, a multiplication operation might be achieved using repeated additions or just bit shifts. POLLUX in its present form cannot detect such semantic similarity, since it leverages side-effects, which could be significantly different for both mechanisms. In future, we plan to augment POLLUX with symbolic execution capabilities to detect such semantic similarities.

- POLLUX's function signature matching is $O(n^2)$, and requires significant computation for comparing graphs with several hundred thousand nodes. We plan to optimize it as part of future work.

8. RELATED WORK

GENERAL PURPOSE PLATFORMS. BitBlaze [46] is a binary analysis platform that leverages static and dynamic analysis techniques, dynamic symbolic execution, and whole-system emulation and binary instrumentation. Phoenix [17] requires debugging information and thus, unlike BitBlaze, is not a binary-only analysis platform. Both BitBlaze and Phoenix can also be used to detect changes to binary dependencies. However, they employ heavy machinery to achieve the desired result. In contrast, POLLUX uses light-weight and robust dynamic binary analysis techniques.

STATIC ANALYSIS. There exist several static binary analysis platforms such as BAP [29], CodeSurfer/x86 [26], and Jakstab [38]. CodeSurfer/x86 and Jakstab first disassemble binary code, reconstruct call and control flow graphs, and then perform static analysis over the reconstructed control flow. BAP lifts the instructions to an intermediate language (IL), and then performs analysis at the IL level. In contrast, POLLUX leverages dynamic binary analysis to develop an execution call graph, and examines it to determine semantic dissimilarities.

DYNAMIC ANALYSIS. POLLUX is most closely related to BLEX [34], which observes the side effects of function execution under a controlled randomized environment. Two functions are deemed similar, if their corresponding side effects are similar. POLLUX also uses the notion of similarity in side-effects, but unlike BLEX, does not require any controlled environment. Like BLEX, POLLUX is also robust to compiler optimizations, but is significantly more light-weight in its approach. Additionally, POLLUX, like Zhang *et al.* [50] and Nagarajan *et al.* [42], augments its call graph analysis with features, such as intermediate values, to fingerprint functions across binaries.

SYMBOLIC ANALYSIS. Several frameworks, such as BinHunt [36], Bouncer [32], BitFuzz [31], FuzzBall [40], and McVeto [48], operate solely on binaries. BinHunt is similar to POLLUX in spirit, and determines semantic differences in binary programs. However, unlike POLLUX, BinHunt detects semantic similarity using control flow analysis using graph isomorphism technique, symbolic execution, and theorem proving mechanisms. Brumley *et al.* [28] use symbolic mechanisms to determine whether different implementations of the same specification are semantically similar or not.

GRAPH ISOMORPHISM. Unlike POLLUX, BinDiff [33, 35] and BinSlayer [27] use graph isomorphism techniques that performs extremely well in both correctness and speed if the two binaries are similar. However, graph isomorphism, in general, does not perform well when the change between two binaries is large.

9. CONCLUSION

We present the design and implementation of POLLUX, a framework that leverages relevant application test cases to drive execution through two versions of the concerned library binary, records all concrete effects on the environment, and compares them to determine semantic similarity for the same API invocation across the two library versions. Our evaluation of POLLUX with 16 open-source libraries confirms its utility, and also indicates both high accuracy and precision even in the face of compiler optimizations.

10. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback.

11. REFERENCES

[1] Capstone. https://github.com/aquynh/capstone.

[2] Catch. https://github.com/philsquared/Catch.

[3] Curl. https://github.com/curl/curl.

[4] DevIL. https://github.com/DentonW/DevIL.

[5] Docker. https://www.docker.com/.

[6] dropbox/json11. https://github.com/dropbox/json11.

[7] Eigen. https://bitbucket.org/eigen/eigen/.

[8] Fit. https://github.com/pfultz2/Fit.

[9] GSL - GNU Scientific Library. https://github.com/ampl/gsl.

[10] HTTP parser. https://github.com/nodejs/http−parser.

[11] libarchive. https://github.com/libarchive/libarchive.

[12] libtorrent. https://github.com/arvidn/libtorrent.

[13] libxml violates the zlib interface and crashes. https://mail.gnome.org/archives/xml/2010−January/msg00035.html.

[14] Onion. https://github.com/davidmoreno/onion.

[15] OpenSSH Patches Critical Flaw That Could Leak Private Crypto Keys. http://www.openssh.com/txt/release−7.1p2.

[16] PEGTL. https://github.com/ColinH/PEGTL.

[17] Phoenix Compiler and Shared Source Common Language Infrastructure. http://research.microsoft.com/en−us/collaboration/focus/cs/phoenix.aspx.

[18] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en−us/articles/pin−a−dynamic−binary−instrumentation−tool.

[19] Pin 2.11 User Guide. https://software.intel.com/sites/landingpage/pintool/docs/49306/Pin/html/.

[20] Sørensen-Dice coefficient. https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient.

[21] spdlog. https://github.com/gabime/spdlog.

[22] The Heartbleed Bug. http://heartbleed.com/.

[23] Valijson. https://github.com/tristanpenman/valijson.

[24] Winamp crashes at launch. http://forums.winamp.com/showthread.php?t=374649.

[25] zlib. https://github.com/madler/zlib.

[26] BALAKRISHNAN, G., GRUIAN, R., REPS, T., AND TEITELBAUM, T. CodeSurfer/x86—A Platform for Analyzing x86 Executables. In *CC'05*.

[27] BOURQUIN, M., KING, A., AND ROBBINS, E. BinSlayer: Accurate Comparison of Binary Executables. In *PPREW '13*.

[28] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *USENIX Security'07*.

[29] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A Binary Analysis Platform. In *CAV'11*.

[30] BUSINGE, J., SEREBRENIK, A., AND VAN DEN BRAND, M. An Empirical Study of the Evolution of Eclipse Third-party Plug-ins. In *IWPSE-EVOL '10*.

[31] CABALLERO, J., POOSANKAM, P., MCCAMANT, S., BABI Ć, D., AND SONG, D. Input Generation via Decomposition and Re-stitching: Finding Bugs in Malware. In *CCS '10*.

[32] COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: Securing Software by Blocking Bad Input. In *SOSP '07*.

[33] DULLIEN, T., AND ROLLES, R. Graph-based comparison of Executable Objects. In *SSTIC '05*.

[34] EGELE, M., WOO, M., CHAPMAN, P., AND BRUMLEY, D. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Security '14*.

[35] FLAKE, H. Structural Comparison of Executable Objects. In *DIMVA '04*.

[36] GAO, D., REITER, M., AND SONG, D. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *ICICS '08*.

[37] KINDER, J. *Static Analysis of x86 Executables*. PhD thesis, TU Darmstadt, 2010.

[38] KINDER, J., AND VEITH, H. Jakstab: A Static Analysis Platform for Binaries. In *CAV '08*.

[39] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05*.

[40] MARTIGNONI, L., MCCAMANT, S., POOSANKAM, P., SONG, D., AND MANIATIS, P. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *ASPLOS '12*.

[41] MILEVA, Y. M., DALLMEIER, V., BURGER, M., AND ZELLER, A. Mining Trends of Library Usage. In *IWPSE-Evol '09*.

[42] NAGARAJANA, V., GUPTA, R., ZHANG, X., MADOU, M., AND DE SUTTER, B. Matching Control Flow of Program Versions. In *ICSM '07*.

[43] RAEMAEKERS, S., VAN DEURSEN, A., AND VISSER, J. Measuring software library stability through historical version analysis. In *ICSM 2012*.

[44] RAEMAEKERS, S., VAN DEURSEN, A., AND VISSER, J. Semantic Versioning Versus Breaking Changes: A Study of the Maven Repository. In *SCAM '14*.

[45] SEO, H., SADOWSKI, C., ELBAUM, S., AFTANDILIAN, E., AND BOWDIDGE, R. Programmers' Build Errors: A Case Study (at Google). In *ICSE 2014*.

[46] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ICISS '08*.

[47] TEYTON, C., FALLERI, J., PALYART, M., AND BLANC, X. A Study of Library Migrations in Java. *Journal of Software: Evolution and Process, 2014*.

[48] THAKUR, A., LIM, J., LAL, A., BURTON, A., DRISCOLL, E., ELDER, M., ANDERSEN, T., AND REPS, T. Directed Proof Generation for Machine Code. In *CAV'10*.

[49] TIFFANY BAO, J. B., AND WOO, M. ByteWeight: Learning to Recognize Functions in Binary Code. In *Security '14*.

[50] ZHANG, F., JHI, Y.-C., WU, D., LIU, P., AND ZHU, S. A First Step Towards Algorithm Plagiarism Detection. In *ISSTA '12*.