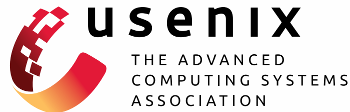


Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components

Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley,
Carnegie Mellon University

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

This paper is included in the Proceedings of the 23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7

Open access to the Proceedings of the 23rd USENIX Security Symposium is sponsored by USENIX

Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components

Manuel Egele
Carnegie Mellon University
megele@cmu.edu

Maverick Woo
Carnegie Mellon University
pooh@cmu.edu

Peter Chapman
Carnegie Mellon University
peter@cmu.edu

David Brumley
Carnegie Mellon University
dbrumley@cmu.edu

Abstract

Matching function binaries—the process of identifying similar functions among binary executables—is a challenge that underlies many security applications such as malware analysis and patch-based exploit generation. Recent work tries to establish semantic similarity based on static analysis methods. Unfortunately, these methods do not perform well if the compared binaries are produced by different compiler toolchains or optimization levels. In this work, we propose *blanket execution*, a novel dynamic equivalence testing primitive that achieves complete coverage by overriding the intended program logic. Blanket execution collects the side effects of functions during execution under a controlled randomized environment. Two functions are deemed similar, if their corresponding side effects, as observed under the same environment, are similar too.

We implement our blanket execution technique in a system called BLEX. We evaluate BLEX rigorously against the state of the art binary comparison tool BinDiff. When comparing optimized and un-optimized executables from the popular GNU coreutils package, BLEX outperforms BinDiff by up to 3.5 times in correctly identifying similar functions. BLEX also outperforms BinDiff if the binaries have been compiled by different compilers. Using the functionality in BLEX, we have also built a binary search engine that identifies similar functions across optimization boundaries. Averaged over all indexed functions, our search engine ranks the correct matches among the top ten results 77% of the time.

1 Introduction

Determining the semantic similarity between two pieces of binary code is a central problem in a number of security settings. For example, in automatic patch-based exploit generation, the attacker is given a pre-patch binary and a post-patch binary with the goal of finding the patched vulnerability [4]. In malware analysis, an analyst is given a number of binary malware samples and wants to find similar malicious functionality. For instance, previous work by Bayer et al. achieves this by clustering the recorded execution behavior of each sample [2]. Indeed, the semantic similarity problem is important enough that the DARPA CyberGenome program has spent over $43M to develop new solutions to it and its related problems [7].

An inherent challenge shared by the above applications is the problem of semantic binary differencing ("diffing") between two binaries. A number of binary diffing tools exist, with current state-of-the-art diffing algorithms such as zynamics BinDiff[1] [8, 9] taking a graph-theoretic approach to finding similarities and differences. BinDiff takes as input two binaries, finds functions, and then performs graph isomorphism (GI) detection on pairs of functions between the binaries. BinDiff highlights pairs of function code blocks between the binaries that are similar and different. Although the graph isomorphism problem has no known polynomial time algorithm, BinDiff has been carefully designed with clever heuristics to make it usably fast in practice. This graph-theoretic approach pioneered by BinDiff has inspired follow-up work such as BinHunt [10] and BinSlayer [3].

While GI-based approaches work well when two semantically equivalent binaries have similar control flow graphs (CFG), it is easy to create semantically equivalent binaries that have radically different CFGs. For example, compiling the same source program with `-O0` and `-O3` radically changes both the number of nodes and structure of edges in both the control flow graph and the call graph. Our experiments show that even this common change to the compiler's optimization level invalidates this assumption and reduces the accuracy of the GI-based BinDiff to 25%.

In this paper, we present a new binary diffing algorithm that does not use GI-based methods and as a result finds similarities where current techniques fail. Our insight is that regardless of the optimization and obfuscation differ-

[1] http://www.zynamics.com/bindiff.html

ences, similar code must still have semantically similar execution behavior, whereas different code must behave differently. At a high level, we execute functions of the two input binaries in tandem with the same inputs and compare observed behaviors for similarity. If observed behaviors are similar across many randomly generated inputs, we gain confidence that they are semantically similar. The main idea of executing programs on many random inputs to test for semantic equivalence is inspired by the problem of polynomial identity testing (PIT). At a high level, the PIT problem seeks efficient algorithms to test if an arithmetic circuit C that computes a polynomial $p(x_1, \cdots, x_n)$ over a given base field $\mathbb{F}$ outputs zero for *every* one of the $|\mathbb{F}|^n$ possible inputs. The earliest algorithm for PIT was a very intuitive randomized algorithm that simply runs C on random inputs. This algorithm depends on the fact that if p is not identically zero, then the probability that C returns zero on a randomly-chosen input is "*small*."[2] By repeating this test, either we will hit an input $(x_1, \cdots, x_n)$ such that $p(x_1, \cdots, x_n) \neq 0$, or we gain confidence that p is identically zero.

There are many challenges to applying the general PIT idea to actual programs, however. Arithmetic circuits have well-defined inputs and outputs, but it is currently an area of active research to identify the inputs and outputs of functions in binary code (see, e.g., [5]). Instead, we propose seven assembly-level features to record during the execution of each function as an approximation of its semantics. Additionally, while it is straightforward to evaluate an arithmetic circuit entirely, finding a collection of inputs that can execute and thus extract the semantics of every part of a program is another open research problem. To achieve full coverage, we repeatedly start execution from the first un-executed instruction of a function until every instruction has been executed at least once.

We have implemented a dynamic equivalence testing system called BLEX to evaluate our blanket execution technique. Our system observes seven semantic features from an execution, namely the four groups of values read from and written to the program heap and stack, the calls made to imported library functions, the system calls made during execution, and the values stored in the `%rax` register upon completion of the analyzed function. We compute the semantic similarity of two functions by taking a weighted sum of the Jaccards of the seven features. Our evaluation is based on a comprehensive dataset. Specifically, we compiled GNU coreutils 8.13 with three current compiler toolchains—`gcc` 4.7.2, `icc` 14.0.0, and `clang` 3.0-6.2. Then, for each compiler toolchain, we compiled coreutils at optimization levels 0 to 3, producing 12 versions of coreutils in total.

[2] The precise upperbound on this probability is commonly known as the Schwartz-Zippel Lemma [23, 28].

Overall, our contributions are as follows:

- We propose *blanket execution*, a novel full-coverage dynamic analysis primitive designed to support semantic feature extraction (§3). Unlike previous approaches such as [25], blanket execution ensures the execution of every instruction without forced violation of branch instruction semantics.
- We propose seven binary code semantics extractors for use in blanket execution. This allows us to approximate the semantics of a function without relying on variable identification or source code access.
- We implement the proposed algorithm in a system called BLEX and evaluate it on a comprehensive dataset based on GNU Coreutils compiled on 4 optimization levels by 3 compilers. Our experiments show that BinDiff performs well (8% better than BLEX) on binaries that are syntactically similar. For binaries that show significant syntactic differences, BLEX outperforms BinDiff by a factor of up to 3.5 and a factor of 2 on average.

2 Problem Setting and Challenges

The problem of matching function binaries is a significant challenge in computer security. In this problem setting, we are only given access to binary code without debug symbols or source. We assume the code is not packed and is compiled from a high-level language that has the notion of a function, i.e., not hand-written assembly. While handling packed code is important, it poses unique challenges which are out of scope for this paper. There are many real-life examples of such problem settings in security. These include, for example, automatic patch-based exploit generation [4], reverse engineering of proprietary code [24], and finding bugs in off-the-shelf software [6].

Clearly, all compiled versions of the same source code should be considered similar by a system addressing the problem of matching function binaries. In this paper, we explicitly consider the case where different compilers and optimization settings produce different binary programs from identical source code. Changing or updating compilers and optimizers happens periodically in industry. For example, with the release of the Xcode 4.1 IDE, Apple switched the default compiler suite from `gcc` to `llvm` [1]. Furthermore, changing compilers and optimization settings is similar to an obfuscation technique. It is common for optimizers to substitute instruction sequences with semantically equivalent but syntactically different sequences. This is exactly a form of metamorphism.

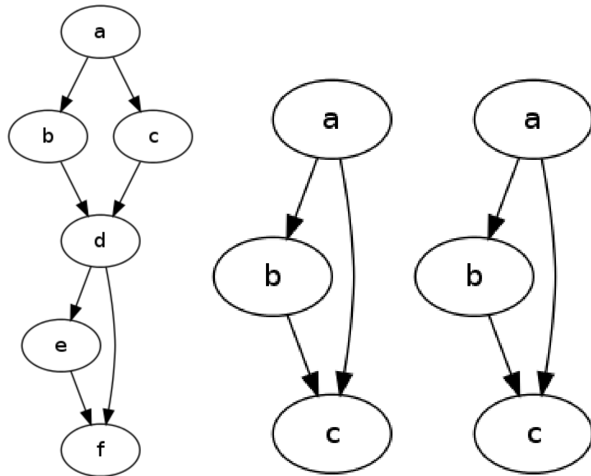
As a motivating example, consider the problem of determining similarities in `ls` compiled with `gcc -O0` and `gcc -O3`, as shown in Figure 1. Although the two assembly listings are the result of the exact same source code, almost all syntactic similarities have been eliminated by the applied compiler optimizations. *If we cannot handle*

```
1    static int strcmp_name(
2        V a, V b
3    )
4    {
5    return cmp_name(a, b, strcmp);
6    }
7
8    static inline int
9    cmp_name (
10     struct fileinfo const *a,
11     struct fileinfo const *b,
12     int (*cmp) (
13         char const *,
14         char const *)
15   )
16   {
17     return
18         cmp (a->name, b->name);
19   }
```

```
1    407ab9 <strcmp_name>:
2       ab9: push %rbp
3    ...
4       ad1: mov $0x402710,%edx
5    ... PLT entry of strcmp
6       ad6: mov %rcx,%rsi
7       ad9: mov %rax,%rdi
8       adc: callq 406fa1 <cmp_name>
9       ae1: leaveq
10      ae2: retq
11
12   406fa1 <cmp_name>:
13      fa1: push %rbp
14   ...
15      fcd: callq *%rax
16   ... call function pointer (e.g., strcmp)
17      fcf: leaveq
18      fd0: retq
```

```
1    4053e0 <strcmp_name>:
2        e0: mov (%rsi),%rsi
3        e3: mov (%rdi),%rdi
4        e6: jmpq 402590 <strcmp@plt>
```

Figure 1: `strcmp_name` from `ls`. Source (left), compiled with `gcc -O0` (center), and `gcc -O3` (right).



(a) md5_finish_ctx (unoptimized) (b) md5_finish_ctx (optimized) (c) xstrxfrm (optimized)

Figure 2: Only the CFG (b), but not (c), is the correct match for (a).

a short function in `coreutils` "obfuscated" only by different optimization levels, what hope do we have on real threats?

The difference between optimized and non-optimized code illustrates several key challenges for correctly identifying the two code sequences as similar:

- Semantically similar functions may not yield syntactically similar binaries. The length of code and operations performed between the two optimization levels is radically different although they both carry out the same simple operation.
- The analysis needs to reason about how memory is read and written. For example, the `-O0` and `-O3` access their arguments identically despite `-O3` not setting up a typical stack frame. In addition, the `cmp_name` function in the `-O0` code up to the call on line 15 indexes struct fields in a semantically equivalent manner to lines 1 and 2 of the `-O3` version.
- Inter-procedural and context sensitive analysis is a must. In `-O0`, `strcmp_name` will always call `cmp_name` with a function pointer pointing to `strcmp`, but in `-O3`, `strcmp` is called directly.

Unfortunately, existing systems both in the security and the general systems community do not address all the above challenges. Syntax-only approaches such as BitShred [12] and others will fail to find any similarities in the code presented in Figure 1. GI-based algorithms will fail because the call and control flow graphs are radically different. GI based methods, such as BinDiff, also face challenges when the control flow graphs to compare are small and collisions render them indistinguishable. Consider, for example the three control flow graphs in Figure 2. The CFG in (a) is the unoptimized version of the `md5_finish_ctx` function in the `sort` utility. While Figure (b) is the optimized version of that function, Figure (c) is the implementation of `xstrxfrm` in the same binary. However, an approach that solely relies on graph similarities, will likely not be able to make a meaningful distinction in this scenario. Alternative approaches, such as the one proposed by Jiang and Su [14] perform only intra-procedural analysis and thus are not able to identify the similarity of the two implementations. To address the above-mentioned challenges in the scope of matching function binaries, we propose a novel dynamic analysis.

3 Approach

We propose *blanket execution* as a novel dynamic analysis primitive for semantic similarity analysis of binary code. Blanket execution of a function f dynamically executes the function repeatedly and ensures that each

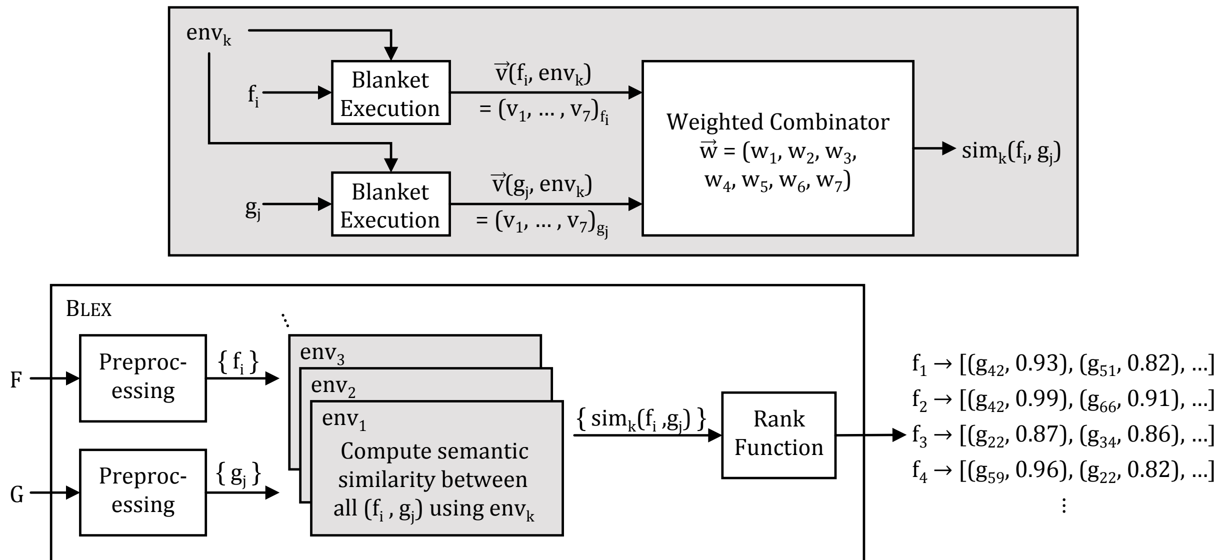


Figure 3: System overview. The upper diagram shows how blanket execution is used to compute the semantic similarity between two given functions f and g inside a given environment env_k. The lower diagram shows how the above computation is used in our BLEX system to, given two program binaries F and G, compute for each function $f_i \in F$ a list of (function, similarity) pairs where each function is a function in G and the list is sorted in non-increasing similarity.

instruction in f is executed at least once. To achieve full coverage, blanket execution starts successive runnings at so-far uncovered instructions. During these repeated runnings, blanket execution monitors and records a variety of dynamic runtime information (i.e., features). Similarity of two functions analyzed by blanket execution is then assessed by the similarity of the corresponding observed features.

More precisely, blanket execution takes a function f and an environment *env* and outputs a vector of dynamic features $\vec{v}(f, env)$ whose coordinates are the feature values captured during the blanket execution. We define the concept of "dynamic feature" broadly to include any information that can be derived from observations made during execution. As an example, we define a feature that corresponds to the set of values read from the heap during a blanket execution.

The novelty of our blanket execution approach lies in (i) *how* the function f is executed for the purpose of feature collection and (ii) *what* features are collected so that they are useful for semantic similarity comparisons. We will first look at (i) while assuming an abstract set of N features in (ii). The latter will be fully specified and explained in §4. For convenience, we will denote each coordinate of a vector $\vec{v}$ as v_i.

3.1 Environment

A key concept in blanket execution is the notion of the *environment* in which a blanket execution occurs. Blanket execution is a dynamic program analysis primitive. This means that in order to analyze a target, blanket execution runs the target and monitors its execution.

To concretely run binary code, we need to provide concrete values of the set of registers and memory locations being read. In blanket execution, we provide concrete initial values to *all* registers and *all* memory locations regardless of whether they are read or not. For unmapped memory regions, an environment also specifies a randomized but fixed *dummy* memory-page. Together, this set of values is known as "an environment." The most important property of an environment is that it must be *efficiently reproducible* since we need to be able to efficiently use a specific environment for multiple runs. This is particularly crucial due to our need to compare feature vectors collected from different functions.

3.2 Blanket Execution

Definition. Given a function f and an environment *env*, the blanket execution of f in *env* is the repeated runnings of f starting from the first un-executed instruction of f until every instruction in f has been executed at least once. Each one of these repeated runnings is called a *blanket execution run* of f, or "be-run" for short. Since we will be using a fixed environment to perform be-runs on a large number of functions, we also define a *blanket execution campaign* ("be-campaign") to be a set of be-runs using the same environment.

Notice that the description of blanket execution encompasses a notion of regaining control. There are several possible outcomes after we start to run f. For example,

f may terminate, f may trigger a system exception, or f may go into an infinite loop. We explain how to handle these possibilities in §4.2.

```
input  : Function binary f, Environment env
output : Feature vector v(f, env) of f in env
I ← getInstructions(f)
fvec ← emptyVector()
while I ≠ ∅ do
    addr ← minAddr(I)
    (covered, v) ← be-run(f, addr, env)
    I ← I \ covered
    fvec ← pointwiseUnion(fvec, v)
end
return fvec
```

Algorithm 1: Blanket Execution.

Algorithm. Algorithm 1 outlines the process of blanket execution for a given function f and an execution environment *env*. First, the function f is dissected into the set of its constituent instructions ($\mathbb{I}$). The system executes the function in the environment *env* at the instruction with the lowest address in $\mathbb{I}$, recording the targeted observations. Executed instructions are removed from $\mathbb{I}$ and the process repeats until all instructions of the function have been executed (i.e., $|\mathbb{I}| = 0$). All recorded feature values, such as memory accesses and system call invocations, are aggregated into a feature vector (*fvec*) associated with the function. Each element in the resulting feature vector is the union of all observed effects for the respective feature.

Rationale. A common weakness of dynamic analysis solutions is potentially low coverage of the program under test. Intuitively, this is because a dynamic analysis must provide an input to drive the execution of the program but by definition a fixed input can exercise only a fixed portion of the program. Although multiple inputs can be used in an attempt to boost coverage, it remains an open research problem to generate inputs to boost coverage effectively. Blanket execution side-steps this challenge and attains full coverage by sacrificing the natural meaning of "executing a function," namely executing from the start of it.

3.3 Assessing Semantic Similarity

The output of a blanket execution on a function f in an environment *env* is a length-N feature vector $\vec{v}(f, env)$. In this section we define how to compute $sim_k(f,g)$, the semantic similarity of two functions f and g given two feature vectors $\vec{v}(f, env)$ and $\vec{v}(g, env)$ that were extracted using blanket execution under the same environment env_k.

Definition. All our features are *sets* of values and we use the Jaccard index to measure the similarity between sets. We define $sim_k(f,g)$ to be a normalized weighted sum of the Jaccard indices on each of the N features in env_k. Mathematically, given N weights $w_1, \ldots, w_N$, we define

$$sim_k(f,g) = \sum_{i=1}^{N} \left(w_i \times \frac{|v_i(f, env_k) \cap v_i(g, env_k)|}{|v_i(f, env_k) \cup v_i(g, env_k)|} \right) / \sum_{\ell=1}^{N} w_\ell .$$

The numerator computes the weighted sum of the Jaccard indices and the denominator computes the normalization constant. The normalization ensures that $sim_k(f,g)$ ranges from 0 to 1, capturing the intuition that it is a similarity measure.

Similarity, Not Equivalence! As explained in §1, our work draws inspiration from the randomized testing algorithm for the polynomial identity testing problem. Strictly speaking, if two functions behave differently in just one environment, we can declare that they are inequivalent. However, in order to make such a judgment, we must have a precise and accurate method to capture the execution behavior of a function. While this is straightforward for arithmetic circuits, it is unsolved for binary code in general. Furthermore, in many applications such as malware analysis, analysts may intend to identify *both* identical and similar functions. This is why we assess the notion of semantic similarity for binary code instead of semantic equivalence.

Weights. Different features may carry different degrees of importance. To allow for this flexibility, we use a weighted sum of the Jaccard indexes. We explain our method to compute the weights (w_ℓ) in §5.1.2.

3.4 Binary Diffing with Blanket Execution

Given the ability to compute the semantic similarity of two functions in a fixed environment, we can perform binary diffing using blanket execution. Figure 3 illustrates the workflow of our system, BLEX.

Preprocessing. Given two binaries F and G, we first preprocess them into their respective sets of constituent functions. We denote these sets as $\{f_i\}$ and $\{g_j\}$ respectively.

Similarity Computation with Multiple Environments. Just as in polynomial identity testing, we will compute the similarity of every pair of (f_i, g_j) in multiple randomized environments $\{env_k\}$. Recall from §3.3 that $sim_k(f_i, g_j)$ is the computed semantic similarity of f_i and g_j in env_k.

Ranking by Averaged Similarity. For each (f_i, g_j), we compute their similarity by averaging over the environments. Let K be the number of environments used. Mathematically, we define

$$sim(f_i, g_j) = \frac{1}{K} \sum_k sim_k(f_i, g_j).$$

Finally, for each function f_i in the given binary F, we output a list of (function, similarity) pairs where each function is an identified function in G and the list is sorted in non-increasing similarity. This completes the process illustrated in Figure 3.

4 Implementation

We implemented the approach proposed in §3 in a system called BLEX. BLEX was implemented and evaluated on Debian GNU/Linux 7.4 (Wheezy) in its amd64 flavor. Because BLEX uses the Pin dynamic instrumentation framework [17] (see §4.2), it is easily portable to other platforms supported by Pin (e.g., Windows or 32-bit Linux).

4.1 Inputs to Blanket Execution

BLEX operates on two inputs. The first input is a program binary F, and the second input is an execution environment under which blanket execution is performed. In a first pre-processing step, BLEX dissects F into individual functions f_i. Subsequently, BLEX applies blanket execution to the f_i as explained in §3. Furthermore, Algorithm 1 uses a static analysis primitive `getInstructions`, which dissects a given function into its constituent instructions.

Reliably identifying function boundaries in binary code is an open research challenge and a comprehensive solution to the function boundary identification problem is outside the scope of this work. However, heuristic approaches, such as Rosenblum et al. [22] or the techniques implemented in IDA Pro [11] deliver reasonable accuracy when identifying function boundaries. IDA Pro supports both primitives used by blanket execution (i.e., function boundary identification and instruction extraction). BLEX thus defers these tasks to the IDA Pro disassembler.

4.2 Performing a BE-Run

A blanket execution run starts execution of a function at a given address i under an environment env. However, given a program binary, one cannot just instruct the operating system to start execution at said address. Upon program startup, the operating system and loader are responsible for mapping the executable image into memory and transferring control to the program entry point defined in the file header. We leverage this insight to correctly load the application into memory. Once the loader transfers control to the program entry point, we divert control to the address from which to perform the blanket execution run (address i). Letting the loader perform its intended operation means that the executable will be loaded with its valid expected memory layout. Note that valid here only means that all sections of the binary are correctly mapped to memory.

Applications frequently make use of functions imported from shared libraries. On Linux the runtime linker implements lazy evaluation of entries in the procedure linkage table (plt). That is, function addresses are only resolved the first time the function is called. However, the side effects produced by the dynamic linker are not characteristic of function behavior. Instead, these side effects create unnecessary noise during blanket execution. To prevent such noise, BLEX sets the `LD_BIND_NOW` environment variable to instruct ld.so (on Linux) to resolve all plt entries at program startup.

Once the be-run starts, BLEX records the side effects produced by the code under analysis. To this end, BLEX leverages the Pin dynamic instrumentation framework to monitor memory accesses and other dynamic execution characteristics, such as system calls and return values (see §4.3). Program code that executes in a random environment is expected to reference unmapped memory. Such invalid memory accesses commonly cause a segmentation fault. To prevent this common failure scenario, BLEX intercepts accesses to unmapped memory. Instead of terminating the analysis, BLEX replaces the referenced (unmapped) memory page with the contents of a dummy memory page specified in the environment. This allows execution to continue without terminating the analysis.

When Does a Run Terminate? A be-run is an interprocedural dynamic analysis of binary code. However, such a dynamic analysis is not guaranteed to always terminate within a reasonable amount of time. In particular, executing under a randomized environment can easily cause a situation where the program gets stuck in an infinite loop. To avoid such situations and guarantee forward progress, BLEX continuously evaluates the following criteria to determine if a be-run is completed.

1. Execution reaches the end of the function in which the be-run started.
2. An exception is raised or a terminal signal is received.
3. A configurable number of instructions have been executed.
4. A configurable timeout has expired.

BLEX detects that a function finished execution by keeping a counter that corresponds to the depth of the call stack. Upon program startup the counter is initialized to zero. Each `call` instruction increments the counter and each `ret` instruction decrements the counter by one. As soon as the counter drops below zero, the be-run is said to be completed.

To catch exceptions and signals, BLEX registers a signal handler and recognizes the end of a be-run if a signal is received. If the code under analysis registered a signal handler for the received signal itself, BLEX does not terminate the be-run but passes the signal on to the appropriate signal handler.

4.3 Instrumentation

Blanket execution monitors the side effects of program execution. A wealth of systems such as debuggers, emulators, and virtual machines have been used in the past to implement dynamic analyses. We chose to implement BLEX on Intel's Pin dynamic instrumentation framework because the tool is mature, well documented, and proven in practice.

At its core, Pin employs a just-in-time (JIT) compiler to recompile basic blocks of binary application code at runtime. Instead of running the original application code, Pin recompiles a block of code, inserting the instrumentation functionality specified by the developer. Then the recompiled code is executed. Upon completion of the code block, Pin regains control and repeats the same process for the next block of application code. The analysis functionality is specified by the developer in a so-called pintool. A pintool is a collection of instrumentation and analysis routines written in C++.

BLEX uses a pintool to instrument individual instructions and record their effects during a be-run. In our implementation we chose features that capture a variety of system level information (e.g., memory accesses), as well as higher level attributes, such as function and system calls. While the current list of features can easily be extended, the following features proved useful for establishing function binary similarity:

1. Values read from the program heap (v_1)
2. Values written to the program heap (v_2)
3. Values read from the program stack (v_3)
4. Values written to the program stack (v_4)
5. Calls to imported library functions via the plt (v_5)
6. System calls made during execution (v_6)
7. Return values stored in the `%rax` register upon completion of the analyzed function (v_7)

Each be-run results in seven sets of observations – one set for each feature. Once all instructions for a function f have been covered, BLEX combines all information pertaining to f in seven sets. That is, given a function f, all observations of v_1 are combined into a single set of values for that function (i.e., f_{v_1}). The same process is repeated for the remaining categories to produce $f_{v_2} \dots f_{v_7}$. The result after blanket execution of a program is the list of functions f_i and the seven sets of observed side effects for each f_i.

Categories $v_1 \dots v_4$ and v_7 record the numeric values used in the respective operations (e.g., values read from memory). Category v_5 records the names of the invoked functions, and v_6 records the system calls invoked. Note that the technique of blanket execution neither defines nor restricts extracted features. BLEX can easily be extended with additional features that help characterize functions. §5 shows that the seven categories of features currently extracted by BLEX are well suited to capture the semantic information of functions.

BLEX relies on the observation that many execution side effects are characteristic of function semantics and thus persist between different compilers and optimizations. While compilers certainly cannot optimize system calls without sacrificing correctness, memory accesses are commonly subject to optimizations. For example, an optimized register allocation scheme can prevent the need for aggressively spilling registers onto the stack. This means that some features are more robust and thus more indicative of function semantics than others. To address these varying degrees of influence, BLEX attributes each feature category with a weight factor ($w_1 \dots w_7$). BLEX leverages support vector machines to establish optimal values for these weights (§5.1.2). We will now discuss how BLEX monitors program execution for side effects.

Memory Accesses. The first four categories of side effects ($v_1 \dots v_4$) are derived from memory accesses. BLEX conforms to a standard memory model where reading a previously written memory cell returns the most recent value written to that cell. While this model is intuitive it only applies to valid (i.e., mapped) memory. In the case that a program tries to access unmapped memory, the operating system will raise an invalid memory reference exception. If a function makes use of global variables, or expects a pointer to mapped memory as one of its formal arguments, normal program execution will initialize such memory properly before the function is called. However, because blanket execution is oblivious to such semantic dependencies, it is common that functions access unmapped memory during blanket execution.

To prevent program termination due to unmapped memory accesses, BLEX simulates that all memory references are valid. To this end, whenever the program tries to access unmapped memory, BLEX interrupts program execution and maps a dummy page in that location. This page is then populated with the data from the dummy page specified in the execution environment.

Pin makes it easy for the developer to emulate data transfers from memory to a register. Thus, a naïve but ineffective approach to simulate that all memory is mapped would be to instrument all instructions that transfer data from memory to a register. For example, the instruction

```
mov (%rsi),%rdx
```

will copy the value pointed to by `%rsi` into register `%rdx`. Unfortunately, Pin's capabilities of intercepting memory accesses are restricted to explicit data transfers such as the above. Instructions with input and output operands that are memory cells cannot be instrumented in the same way. For example, the instruction

```
addl $0x1, $0x20(%rax)
```

will add the constant value one to the value that is stored at offset 0x20 from the memory address in `%rax`. Because Pin's instrumentation capabilities are not fine-grained enough to modify the values retrieved during operand resolution, the straight-forward approach to emulate memory accesses is not generally applicable.

Of course, BLEX needs to collect observations from all instructions that access memory and not just those that explicitly transfer values from memory to the register file or vice versa. Thus, BLEX implements the following mechanisms depending on whether an instruction reads, writes, or reads and writes memory.

Read Accesses. The Pin API allows us to selectively instrument instructions that read from memory. Furthermore, Pin calculates the effective address that is used for each memory accessing instruction. Thus, before an instruction reads from memory, BLEX will verify that the effective address that will be accessed by the instruction belongs to a mapped memory page. If no page is mapped at that address, BLEX will map a valid dummy memory page at the corresponding address[3] and the memory access will succeed.

Recall that a blanket execution environment consists of register values and a memory page worth of data that is kept consistent across all blanket execution runs for a given campaign. By seeding dummy pages with the contents specified in the environment, functions that access unmapped memory will read a consistent value. The rationale is that binary code calculates memory addresses either from arguments or global symbols. Similar functions are expected to perform the same arithmetic operations on these values to derive the memory address to access. Consider, for example, the binary implementations illustrated in Figure 1. Both implementations of `strcmp_name` expect and dereference two pointers to `fileinfo` structures (passed in `%rsi` and `%rdi`). During blanket execution these arguments contain random but consistent values as determined by the execution environment. Dereferencing these random values will likely result in a memory access to unmapped memory. By mapping the dummy page at the unmapped memory region, BLEX ensures that both implementations retrieve the same random value from the dummy page.

With this mechanism in place, BLEX can monitor all read accesses to memory by first making sure that the target memory page is mapped, and then read the original value stored at the effective address in memory.

Write Accesses. Similar to read accesses Pin provides mechanisms to instrument instructions that write to memory. However, the Pin API is not expressive enough to record the values that are written to memory. Thus, to record values that are written to memory, BLEX reads the value from memory after the instruction executed. Similar to the read access technique mentioned above, BLEX will make sure that memory writes succeed by mapping a dummy page at the target address if that address resides in unmapped memory.

[3] More precisely, the dummy page is mapped at the target address rounded down to a page-aligned starting address.

Memory Exceptions. BLEX only creates dummy pages for memory accesses to otherwise unmapped memory ranges. If the program tries to access mapped memory in a way that is incompatible with the memory's page protection settings BLEX does not intervene and the operating system raises a segmentation fault. This would occur, for example, if an instruction tries to write to the read-only `.text` section of the program image.

System Calls. Besides memory accesses BLEX also considers the invocation of system calls as side effects of program execution. Pin provides the necessary functionality to intercept and record system calls before they are invoked.

Library Calls. System calls are a well defined interface between kernel space and user space. Thus, they present a natural vantage point to monitor execution for side effects. However, many functions (39% in our experiments) do not result in system calls and thus, relying solely on system calls to identify similar functions is insufficient. Therefore, BLEX also monitors what library functions an application invokes. To support dynamic linking, ELF files contain a `.plt` (procedure linkage table) section. The .plt section contains information (i.e., one entry per function) about shared functions that might be called by the application at runtime. While stripped binaries are devoid of symbol names, they still contain the names of library function names in the plt entries. BLEX records the names of all functions that are invoked via the plt.

While there is no alternative for a program to making system calls, it is not mandatory that shared library functions are invoked through the plt. For example, a developer can chose to statically link a library into her application or interface with the dynamic linker and loader directly by means of the `dlopen` and `dlsym` APIs. Thus, functions from a statically linked version of a program and those from a dynamically linked version thereof will differ in the side effects observed for category library function calls (i.e., v_5).

4.4 Calculating Function Similarity

BLEX combines all of the above methods into a single pintool of 1,036 lines of C++ code. During execution, the pintool collects all necessary information pertaining to the seven observed features. Each be-run results in a feature vector consisting of seven sets that capture the observed side effects. Once all be-runs for a single function finish, BLEX combines the recorded feature vectors and

associates this information with the function. Because the individual dimensions in the vectors are sets, BLEX uses the set-union operation to combine the individual feature vectors, one dimension at a time. As discussed in §3.3, BLEX assesses the similarity of two functions f and g by calculating the weighted sum of the Jaccard indices of the seven dimensions in the respective feature vectors. We use the Jaccard index as a measure of similarity, because even semantically equivalent functions can result in slight differences in the observed feature values. For example, the unoptimized version in Figure 1 will write and read the passed arguments to the stack, whereas the optimized version does not contain such code. This different behavior results in slightly different values of the corresponding coordinates in the feature vectors.

5 Evaluation

BLEX is an implementation of the blanket execution approach to perform function similarity testing on binary programs. We evaluate BLEX to answer the following questions:

- Can BLEX recognize the similarity between semantically similar, yet syntactically different implementations of the same function? (§5.3)
- Can BLEX match functions compiled from the same source code but with different compiler toolchains and/or configurations? (§5.4)
- Is BLEX an improvement over the industry standard tool, BinDiff? (§5.4)
- Can BLEX be used as the basis for high-level applications? (§5.5)

We begin our evaluation with an experiment on syntactically different implementations of the `libc` function `ffs`, followed by an evaluation of the effectiveness of BLEX over BinDiff across a large set of programs with different compilers and compiler configurations, finishing with a prototype search engine for binary programs built on BLEX. Before presenting our results, we discuss the dataset, ground truth, and feature weights used in the evaluation.

5.1 Dataset

For this evaluation we compiled a dataset based on the popular coreutils-8.13 suite of programs. This version of the coreutils suite consists of 103 utilities. However, to prevent damage to our analysis environment, we excluded potentially destructive utilities such as `rm` or `dd` from the dataset, reducing the number of utilities from 103 to 95. We used three different compilers (gcc 4.7.2, icc 14.0.0, and clang 3.0-6.2) with four different optimization settings (`-O0`, `-O1`, `-O2`, and `-O3`) each to create 12 versions of the coreutils suite for the x86-64 architecture. In total our dataset consists of 1,140 unique binary applications, comprising 195,560 functions.

Feature	Accuracy
Read from heap (v_1)	40%
Write to heap (v_2)	57%
Read from stack (v_3)	58%
Write to heap (v_4)	53%
Library function invocation (v_5)	17%
System calls (v_6)	39%
Function return value (v_7)	13%

Table 1: Accuracy of individual features.

5.1.1 Ground Truth

Although BLEX does not rely on or use debug symbols, we compiled all binaries with the `-g` debug flag to establish ground truth based on the symbol names. For our problem setting, we strip all binaries before processing them with BLEX or BinDiff.

Function inlining has the effect that the inlined function disappears from the target binary. Interestingly, the linker can have the opposite effect when it sometimes introduces duplicate function implementations. For example, when compiling the `du` utility, the linker will include five identical versions of the `mbuiter_multi_next` function in the application binary. While such behavior could be explained if the compiler performed code locality optimization, this also happens if all optimization is turned off (`-O0`). This observation suggests that optimization is not the reason for this code duplication. Because these duplicates are exactly identical, we have to account for this ambiguity when establishing ground truth. That is, matching any of the duplicate instances of the same function should be treated equal and correct. In our dataset 37 different programs contained duplicates (between two and six copies) of 16 different functions. Based on these observations, we establish ground truth by considering functions equivalent if they share the same function name.

5.1.2 Determining Optimal Weights

Each feature in BLEX has a weight factor associated with it, i.e., $w_\ell | \ell = 1 \ldots 7$. To assess the sensitivity of BLEX to these weights, we performed seven small-scale experiments as a sensitivity analysis of the individual features. In each experiment, we set all but one weight to zero and evaluated the accuracy of the system when matching functions between all coreutils compiled with `gcc` and the `-O2` and `-O3` optimization settings. Table 1 illustrates how well the individual features BLEX collects can be used to assess similarity between functions.

To establish the optimal values for these weights, we leveraged the Weka[4] (version 3.6.9) machine learning toolkit. Weka provides an implementation of the sequen-

[4] http://www.cs.waikato.ac.nz/ml/weka/

tial minimal optimization algorithm [20] to train a support vector machine based on a labeled training dataset. To train a support vector machine, the training dataset must consist of feature values for positive and negative examples. We created the dataset based on our ground truth by first selecting 9,000 functions at random from our pool of functions. For each function f in a binary F we calculated the Jaccard index with its correct match g in binary G, constituting a positively labeled sample. For each positively labeled sample, we created a negatively labeled sample by calculating the Jaccard index with the feature vector of a random function $g' \in G$ such that $g' \neq g$. The support vector machine determined the weights as $w_2 = 2.4979$, $w_6 = 0.8775$, $w_4 = 0.4052$, $w_1 = 0.3846$, $w_3 = 0.3786$, $w_7 = 0.3222$, and $w_5 = 0.1082$. Using these weights in BLEX to evaluate the dataset from the above-mentioned sensitivity analysis improved accuracy to 75%.

5.2 Experimental Setup

We evaluated BLEX on a commodity desktop system equipped with an Intel Core i7-3770 CPU (4 physical cores @ 3.4GHz) running Debian Wheezy. For this evaluation we set the maximum number of instructions t_i to 10,000 instructions and the timeout for a single be-run to three seconds. We performed blanket execution for all 195,560 functions in our dataset under eleven different environments. On average, 1,590,773 be-runs were required to cover all instructions in the dataset for a total of 17,498,507 be-runs. A single be-run took on average 0.28 seconds, an order of magnitude below the timeout threshold we selected. Only 9,756 be-runs were terminated because of this timeout. 604,491 be-runs (3.5%) were terminated because the number of instructions exceeded the chosen threshold of 10,000 instructions. While performing blanket execution on all 1,140 unique binaries in our dataset required approximately 57 CPU days, performing blanket execution on two versions of the `ls` utility can be achieved in 30 CPU minutes. Because the repeated runnings in blanket execution are independent of each other, blanket execution resembles an embarrassingly parallel workload and scales almost linear with the number of available CPU cores.

5.3 Comparing Semantically Equivalent Implementations

BLEX tracks the observable behavior of function executions to identify semantic similarity independent of the source code implementation. To test our design, we acquired two different implementations of the `ffs` function from the `Newlib` and `uclibc` libraries as used in the evaluation of the system built by Ramos et al. [21] to measure function equivalence in C source code. We compiled both sources with `gcc -O2`. The resulting binaries differed significantly: the control flow graph in the `uclibc` implementation consisted of eleven basic blocks and the `Newlib` implementation consisted of just four basic blocks. We ran BLEX on both function binaries in 13 different random environments. After comparing the resulting feature vectors, BLEX reported perfect similarity between the compiled functions. This result illustrates how BLEX and blanket execution can identify function similarity despite completely different source implementations.

5.4 Function Similarity across Compiler Configurations

The ideal function similarity testing system can identify semantically similar functions regardless of the compiler, optimizations, and even obfuscation techniques employed. The task is nontrivial as different compiler options can result in drastically different executables (see Figure 1). A rough measure of these differences is the number of enabled compiler optimizations. Consider, for example, the number of optimizations enabled by the four common optimization levels in `gcc`. The switch `-O0` turns off all optimization, and `-O1` enables a total of 31 different optimization strategies. Additionally, `-O2` enables another 26 settings, and `-O3` finally adds another nine optimizations. We would expect that binaries compiled from the same source with `-O2` and `-O3` optimizations are closest in similarity. Thus, similarity testing should yield better results for such similar implementations then for binaries compiled with `-O0` and `-O3` optimizations.

We leverage our dataset to compare the accuracy of BLEX and BinDiff in identifying similar functions of the same program, built with different compilers and different compilation options.

Comparison with BinDiff. BinDiff is a proprietary software product that maps similar functions in two executables to each other. To this end, BinDiff assigns a signature to each function. Function signatures initially consist of the number of basic blocks, the number of control flow edges between basic blocks, and the number of calls to other functions. BinDiff immediately matches function signatures that are identical and unique. For the remaining functions, BinDiff applies secondary algorithms, including more expensive graph analyses. One such secondary algorithm matches function names from debug symbols. However, our experiments do not leverage debugging symbols as our efforts are focused on the performance on stripped binaries. The data presented in this evaluation was obtained with BinDiff version 4.0.1 and the default configuration.

As Figure 4 illustrates, BinDiff is very proficient in matching functions among the same utility compiled with the very similar `-O2` and `-O3` settings. Although BLEX also performs reasonably well, BinDiff outperforms BLEX on almost all utilities in this comparison.

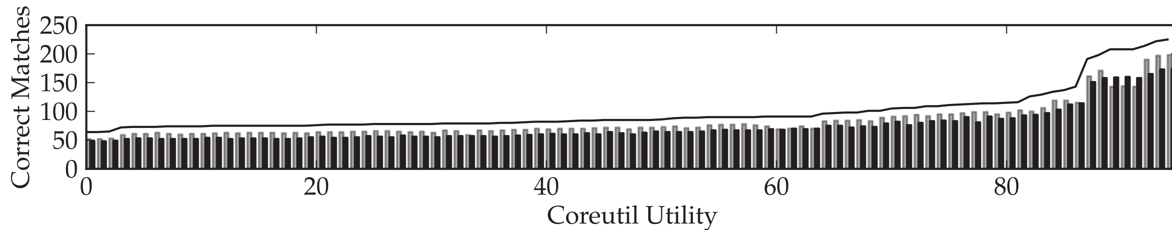


Figure 4: Correctly matched functions for binaries in `coreutils` compiled with `gcc -O2` and `gcc -O3`. BinDiff (grey), BLEX (black), total number of functions in utility (solid line).

The solid line in the figure marks the total number of functions in each utility.

Once the differences between two binaries become more pronounced, BLEX shows considerably improved performance over BinDiff. Figure 5 compares BLEX and BinDiff in identifying similar functions in binaries compiled with the `-O0` and `-O3` optimization settings. This combination of compiler options is expected to produce the least similar binaries and thus should establish a lower bound of the performance one can expect from BLEX and BinDiff respectively. This evaluation shows that BLEX consistently outperforms BinDiff, on average by a factor of two. Furthermore, BLEX matches over three times as many functions correctly for the `du`, `dir`, `vdir`, `ls`, and `chcon` utilities.

Finally, we assess the performance of BLEX and BinDiff on programs built with different compilers. Figure 6 shows the accuracy for binaries compiled with `gcc -O0` and Intel's `icc -O3`. Again, due to the substantial differences in the produced binaries, BLEX consistently outperforms BinDiff in the cross-compiler setting.

Discriminatory power of the similarity score. We also evaluated how well the similarity score tells correct from incorrect matches apart. Similarity scores are normalized to the interval [0,1] with 1 indicating perfect similarity and 0 absolute dissimilarity. In Figure 8, we illustrate the expected similarity value over 10,000 pairs of random functions. On average this expected similarity is 0.12. However, when analyzing the similarity scores of correct matches from the experiment used for Figure 4 (i.e., `gcc -O2` vs. `gcc -O3`), the average similarity score is 0.85. This indicates that the seven features BLEX uses to assess function similarity are indeed suitable to perform this task.

Effects of Multiple Environments. As discussed in §3.4, we proposed to perform blanket execution with multiple environments ($\{env_k\}$). To assess the effects of performing blanket execution under multiple environments, we evaluated how the percentage of correct matches varies as k (the number of environment) increases. Our result is shown in Figure 7. The figure shows that a mild increase (from 50% to 55%) in accuracy up until three environments are used. Interestingly, using more than three environments does *not* significantly improve the accuracy of BLEX. This is in stark contrast to the PIT theory. However, as discussed previously, real-world function binaries are not polynomials and BLEX cannot precisely identify all input and output dependencies of a function. Thus, it may not be surprising that a larger number of random environments does not significantly improve the accuracy of the system. We plan to evaluate alternate strategies for crafting execution environments in a "smart" way in the future.

5.5 BLEX as a Search Engine

Matching function binaries is an important primitive for many higher-level applications. To explore the potential of BLEX as a system building-block, we built a prototype search engine for function binaries. Given a search query (a function *f*) and a corpus of program binaries, we can use BLEX to find the program most likely to contain an implementation of *f*. Phrased differently, an analyst presented with an unknown function can search for similar functions encountered in the past. The analyst can then easily apply the knowledge gathered during the previous analysis of similar functions, reducing the time and effort spent on redundant analysis. Similarly, if a match is found in a program for which the analyst has access to debug symbols, the analyst can leverage this valuable information to speed up the analysis of the target function.

To evaluate this application, we chose 1,000 functions at random from the applications compiled with `gcc -O0`. These functions serve as the search queries. We compiled the corpus from programs in coreutils built with `gcc -O1`,`-O2`, and `-O3` respectively (29,015 functions in total). Our prototype search engine ranked the correct match as the first result in 64% of all queries. 77% of the queries were ranked under the first 10 results (e.g., the first page of search results) and 87% were ranked under the first 10 pages of results (i.e., top 100 ranks). Figure 9 depicts this information as the left-hand side of the CDF.

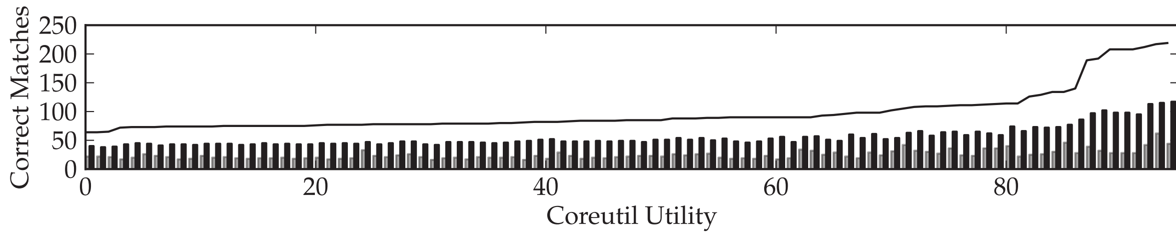


Figure 5: Correctly matched functions for binaries in `coreutils` compiled with `gcc -O0` and `gcc -O3`. BinDiff (grey), BLEX (black), total number of functions in utility (solid line).

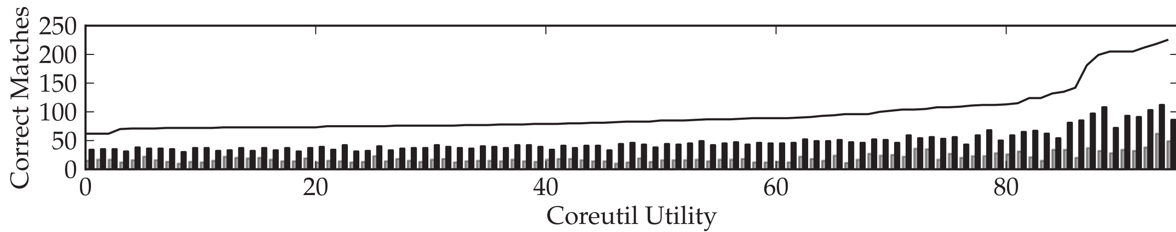


Figure 6: Correctly matched functions for binaries in `coreutils` compiled with `gcc -O0` and `icc -O3`. BinDiff (grey), BLEX (black), total number of functions in utility (solid line).

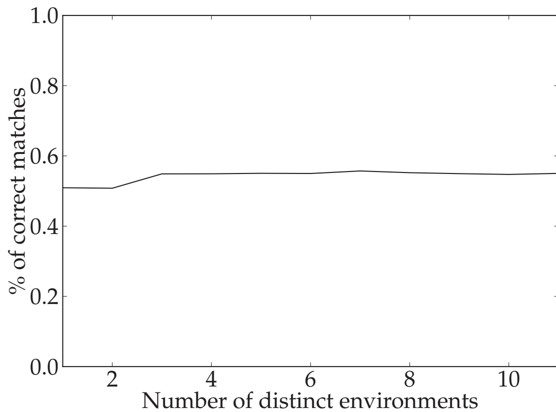


Figure 7: Matching accuracy depending on the number of used environments.

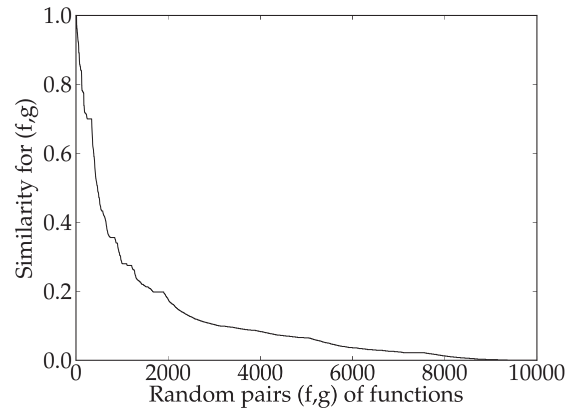


Figure 8: Distribution of similarity scores among 10,000 random pairs of functions. All compiled with `gcc -O2`.

The remaining 13% form a long tail distribution with the worst match at rank 23,261.

The usability of a search engine also depends on its query performance. Our unoptimized implementation answers search queries to the indexed corpus of size 29,015 in under one second on average.

6 Related Work

The problem of testing whether two pieces of syntactically-different code are semantically identical has received much attention by previous researchers. Notably, Jiang and Su [14] recognized the close resemblance of this problem to polynomial identity testing and applied the idea of random testing to automatically mine semantically-equivalent code fragments from a large source codebase. Whereas their definition of semantic equivalence includes only the input-output values of a code fragment and does not consider the intermediate values, we include intermediate values in our features as a pragmatic way to cope with the difficult problem of identifying input-output variables in binary code. Interested readers can see [5, 16] for some of the recent works on that problem.

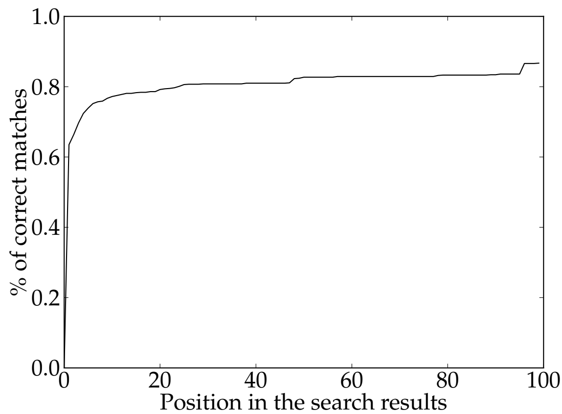


Figure 9: Left-most section of the CDF of ranks for correct matches in 1,000 random search queries.

Intermediate values can also be extremely valuable for other applications. For example, Zhang et al. [26] have investigated how to detect software plagiarism using the dynamic technique of value sequences. This uses the concept of core values proposed by Jhi et al. [13]. The idea is that certain specific intermediate values are unavoidable during the execution of any implementation of an algorithm and are thus good candidates for fingerprinting.

Intermediate values are also used by Zhang and Gupta [27] as a first step in matching the instructions in the dynamic histories of program executions of two program versions. After identifying potential matches as such, Zhang and Gupta refined the match by matching the data dependence structure of the matched instructions. They reported high accuracy in their evaluation using histories from unoptimized and optimized binaries compiled from the same source. This work was used by Nagarajan et al. [18] as the second step of their dynamic control flow matching system. The system by Nagarajan et al. also match functions between unoptimized and optimized binaries. Their technique is based on matching the structure of two dynamic call graphs.

We choose to evaluate BLEX against BinDiff [8, 9] due to its wide availability and also its reputation of being the industry standard for binary diffing. At a high-level, BinDiff starts by recovering the control flow graphs (CFGs) of the two binaries and then attempts to use a heuristic to normalize and match the vertices from the two graphs. Although in essence BinDiff is solving a variant of the graph isomorphism problem of which no efficient polynomial time algorithm is known, the authors of BinDiff have devised a clever neighborhood-growing algorithm that performs extremely well in both correctness and speed if the two binaries are similar. However, as we have explained in the paper, changing the compiler optimization level alone is sufficient to introduce changes that are large enough to confound the BinDiff algorithm.

A noteworthy successor to BinDiff is the BinHunt system introduced in [10]. This paper makes two important contributions. First, it formalized the underlying problem of binary diffing as the Maximum Common Induced Subgraph Isomorphism problem. This allowed the authors to formally and accurately state their backtracking algorithm. Second, instead of relying on heuristics to match vertices and tolerating potential false matches, BinHunt deployed rigorous symbolic execution and theorem proving techniques to *prove* that two basic blocks are in fact equivalent. Unfortunately, BinHunt has only been evaluated in three case studies, all of which support only differences due to patching vulnerabilities. In particular, it has not been evaluated whether BinHunt will perform well on binaries that are compiled with different compiler toolchains or different optimization levels.

A recent addition to this line of work is the BinSlayer system [3]. The authors of BinSlayer correctly observed that graph-isomorphism based algorithms may not perform well when the change between two binaries are large. To alleviate this problem, the authors modeled the binary diffing problem as a bipartite graph matching problem. At a high level, this means assigning a distance between two basic blocks and then pick an assignment (a matching) that maps each basic block from one function to a basic block in another function that *minimizes* the total distance. Among other experiments, the authors evaluated their algorithms by diffing GNU coreutils 6.10 vs. 8.19 (large gap) and also 8.15 vs. 8.19 (small gap). Just as the authors suspected, they observed that graph-isomorphism based algorithms are less accurate in the large-gap experiment than in the small-gap experiment.

Besides binary diffing, our work can also be seen in the light of a binary search engine. Two recent work in the area are Exposé [19] and Rendezvous [15]. Both of these systems are based on static analysis techniques; in contrast, our system is based on dynamic analysis. None of these systems has been evaluated with a dataset that varies both compiler toolchain and optimization level simultaneously.

Finally, semantic similarity can also be used for clustering. For example, Bayer et al. [2] have used ANUBIS for clustering malware based on their recorded behavior. However, this relies on attaining high coverage so that malicious functionality is exposed [25]. We believe that BLEX may also be used for malware clustering.

7 Conclusion

Existing binary diffing systems such as BinDiff approach the challenge of function binary matching from a purely static perspective. It has not been thoroughly evaluated on binaries produced with different compiler toolchains

or optimization levels. Our experiments indicate that its performance drops significantly if different compiler toolchains or aggressive optimization levels are involved.

In this work, we approach the problem of matching function binaries with a dynamic similarity testing system based on the novel technique of blanket execution. BLEX, our implementation of this technique proved to be more resilient against changes in the compiler toolchain and optimization levels than BinDiff.

Acknowledgment

This material is based upon work supported by Lockheed Martin and DARPA under the Cyber Genome Project grant FA975010C0170. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Lockheed Martin or DARPA. This material is further based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 0946825.

References

[1] New Features in Xcode 4.1. https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/WhatsNewXcode/Articles/xcode_4_1.html. Page checked 7/8/2014.

[2] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Network and Distributed System Security Symposium* (2009), The Internet Society.

[3] BOURQUIN, M., KING, A., AND ROBBINS, E. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM Program Protection and Reverse Engineering Workshop* (2013), ACM.

[4] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008), IEEE, pp. 143–157.

[5] CABALLERO, J., JOHNSON, N. M., MCCAMANT, S., AND SONG, D. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Network and Distributed System Security Symposium* (2010), The Internet Society.

[6] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 380–394.

[7] DARPA-BAA-10-36, Cyber Genome Program. https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-36/listing.html. Page checked 7/8/2014.

[8] DULLIEN, T., AND ROLLES, R. Graph-based comparison of executable objects. In *Actes de la Symposium sur la Sécurité des Technologies de l'Information et des Communications* (2005).

[9] FLAKE, H. Structural comparison of executable objects. In *Proceedings of the 2004 Workshop on Detection of Intrusions and Malware & Vulnerability Assessment* (2004), IEEE, pp. 161–173.

[10] GAO, D., REITER, M. K., AND SONG, D. BinHunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security* (2008), Springer, pp. 238–255.

[11] HEX-RAYS. The IDA Pro interactive dissassembler. https://hex-rays.com/products/ida/index.shtml.

[12] JANG, J., BRUMLEY, D., AND VENKATARAMAN, S. BitShred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), ACM, pp. 309–320.

[13] JHI, Y.-C., WANG, X., JIA, X., ZHU, S., LIU, P., AND WU, D. Value-based program characterization and its application to software plagiarism detection. In *Proceeding of the 33rd International Conference on Software Engineering* (2011), ACM, pp. 756–765.

[14] JIANG, L., AND SU, Z. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis* (2009), ACM, pp. 81–92.

[15] KHOO, W. M., MYCROFT, A., AND ANDERSON, R. Rendezvous: A search engine for binary code. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories* (2013), IEEE, pp. 329–338.

[16] LEE, J., AVGERINOS, T., AND BRUMLEY, D. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Network and Distributed System Security Symposium* (2011), The Internet Society.

[17] 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 *Programming Language Design and Implementation* (2005), ACM, pp. 190–200.

[18] NAGARAJAN, V., GUPTA, R., ZHANG, X., MADOU, M., DE SUTTER, B., AND DE BOSSCHERE, K. Matching control flow of program versions. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance* (2007), pp. 84–93.

[19] NG, B. H., AND PRAKASH, A. Exposé: Discovering potential binary code re-use. In *Proceedings of the 37th IEEE Computer Software and Applications Conference* (2013), pp. 492–501.

[20] PLATT, J. C. Sequential Minimal Optimization: A fast algorithm for training Support Vector Machines. Tech. rep., Microsoft Research, 1998.

[21] RAMOS, D. A., AND ENGLER, D. R. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (2011), Springer, pp. 669–685.

[22] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence* (2008), AAAI, pp. 798–804.

[23] SCHWARTZ, J. T. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM 27*, 4 (1980), 701–717.

[24] VAN EMMERIK, M. J., AND WADDINGTON, T. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering* (2004), IEEE, pp. 27–36.

[25] WILHELM, J., AND CHIUEH, T.-C. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection* (2007), Springer, pp. 219–235.

[26] ZHANG, F., JHI, Y.-C., WU, D., LIU, P., AND ZHU, S. A first step towards algorithm plagiarism detection. In *Proceedings of 2012 the International Symposium on Software Testing and Analysis* (2012), ACM, pp. 111–121.

[27] ZHANG, X., AND GUPTA, R. Matching execution histories of program versions. In *Proceedings of the 10th European Software Engineering Conference* (2005), ACM, pp. 197–206.

[28] ZIPPEL, R. Probabilistic algorithms for sparse polynomials. In *Proceedings of the 1979 International Symposium on Symbolic and Algebraic Manipulation* (1979), Springer, pp. 216–226.