Statistical Similarity of Binaries

Yaniv David
Technion, Israel
yanivd@cs.technion.ac.il

Nimrod Partush
Technion, Israel
nimi@cs.technion.ac.il

Eran Yahav
Technion, Israel
yahave@cs.technion.ac.il

Abstract

We address the problem of finding similar procedures in stripped binaries. We present a new statistical approach for measuring the similarity between two procedures. Our notion of similarity allows us to find similar code even when it has been compiled using different compilers, or has been modified. The main idea is to use *similarity by composition*: decompose the code into smaller comparable fragments, define *semantic* similarity between fragments, and use statistical reasoning to lift fragment similarity into similarity between procedures. We have implemented our approach in a tool called `Esh`, and applied it to find various prominent vulnerabilities across compilers and versions, including `Heartbleed`, `Shellshock` and `Venom`. We show that `Esh` produces high accuracy results, with few to no false positives – a crucial factor in the scenario of vulnerability search in stripped binaries.

Categories and Subject Descriptors D.3.4 [Processors: compilers, code generation]; F.3.2 (D.3.1) [Semantics of Programming Languages: Program analysis];

Keywords static binary analysis; verification-aided similarity; partial equivalence; statistical similarity

1. Introduction

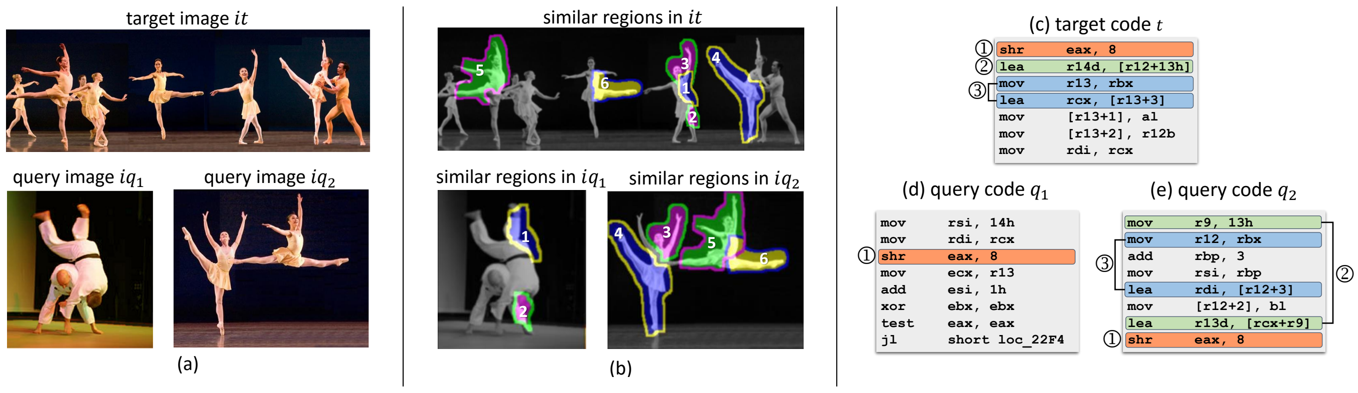
During December 2014, several vulnerabilities were discovered in the prominent implementation of the network time protocol (NTP) `ntpd` [1]. As this implementation is the de facto standard, many products from major vendors were affected, including RedHat's Linux distribution, Apple's OSX and Cisco's 5900x switches. Because some of these vulnerabilities were introduced many years ago, different versions have been ported and integrated into many software packages, ready to be deployed in *binary form* to home computers, enterprise servers and even appliance firmware.

A security savvy individual, or more commonly a security aware company, would want to use a *sample* of the vulnerable product (in binary form) to search for the vulnerability *across all the software* installed in the organization, where source-code is mostly not available. Unfortunately, automatically identifying these software packages is extremely challenging. For example, given a sample from the Debian Linux distribution, finding other (recent or older) vulnerable versions of it is already hard. This is because even though older distributions were probably compiled with the same compiler, `gcc`, they used older `gcc` versions, producing syntactically different binary code. Trying to find OSX applications is even harder, as they are commonly compiled with a different compiler (`CLang`), and the firmware of an appliance, using Intel chips, might be compiled with `icc` (the Intel C compiler), such that the resulting binary procedures differ vastly in syntax. We address this challenge by providing an effective means of searching for semantically similar procedures, at assembly code level.

Problem definition Given a *query* procedure q and a large collection T of (target) procedures, in binary form, our goal is to quantitatively define the similarity of each procedure $t \in T$ to the query q. The main challenge is to define a *semantic* notion of similarity that is precise enough to avoid false positives, but is flexible enough to allow finding the code in *any combination* of the following scenarios: (i) the code was compiled using different compiler versions; (ii) the code was compiled using different compiler vendors; and (iii) a different version of the code was compiled (e.g. a patch). We require a method that can operate without information about the source code and/or tool-chain used in the creation of the binaries *only* of procedures in binary form.

Existing techniques Previous work on clone detection in binaries [29] can overcome different instruction selections, but is unable to handle syntactic differences beyond single instructions. Binary code search [12] is mostly syntactic and therefore fails to handle differences that result from different compilers. Equivalence checking and semantic differencing techniques [15, 23, 24, 27] operate at the source-code

PLDI '16, June 13 - 17, 2016, Santa Barbara, CA, USA
Copyright © 2016 ACM 978-1-4503-4261-2/16/06. . . $15.00
DOI: http://dx.doi.org/10.1145/2908080.2908126



(c) target code t

```
①  shr     eax, 8
②  lea     r14d, [r12+13h]
③┌ mov     r13, rbx
  └ lea     rcx, [r13+3]
    mov     [r13+1], al
    mov     [r13+2], r12b
    mov     rdi, rcx
```

(d) query code q_1

```
    mov     rsi, 14h
    mov     rdi, rcx
①  shr     eax, 8
    mov     ecx, r13
    add     esi, 1h
    xor     ebx, ebx
    test    eax, eax
    jl      short loc_22F4
```

(e) query code q_2

```
     mov     r9, 13h          ┐
  ┌ mov     r12, rbx          │
  │ add     rbp, 3            │
③│ mov     rsi, rbp          │②
  └ lea     rdi, [r12+3]      │
     mov     [r12+2], bl       │
     lea     r13d, [rcx+r9]   ┘
①  shr     eax, 8
```

Figure 1. Image vs. Code Similarity by Composition. Query image iq_2 is similar to the target image it. Query code q_2 is similar to the target code t. Images courtesy of Irani et al. [10]

level and require a matching (labeling of variables) to operate. Furthermore, they do not provide quantitative measures of similarity when procedures are not equivalent. Dynamic equivalence checking techniques [27] are hard to apply as they require obtaining sufficient coverage for each procedure. A dynamic approach for binary code search [13] coerces execution using a randomized environment to achieve coverage, but suffers from a high false positive rate, and does not address or evaluate patching. Data-driven equivalence checking [30] is aimed at translation validation and is limited to short segments of assembly code.

Similarity by composition We draw inspiration from Boiman and Irani's [10] work on image similarity, where the key idea is that one image is similar to another if it can be *composed using regions of the other image*, and that this similarity can be quantified using *statistical reasoning*.

Fig. 1(a) and (b) illustrate the idea of similarity by composition for images. Looking at the three images in Fig. 1(a), we wish to determine which of the two query images, iq_1 and iq_2, is more similar to the target image, it. Our intuition is that iq_2 is probably more similar to it.

Fig. 1(b) provides some explanation for this intuition. Looking at this figure, we identify similar regions in iq_2 and it (marked by the numbered outlined segments). Although the images are not identical, their similarity stems from the fact that query image iq_2 can be composed using significant (and maybe transformed) regions from target image it. In contrast, image iq_1 shares only two small regions with it, and indeed we intuitively consider it to be different. These vague notions are made precise within a statistical framework that lifts similarity between significant regions into similarity between images.

In this paper, we show that the general framework of "similarity by composition" can also be applied to code. Specifically, we consider assembly code extracted from stripped binaries (with no debug information). Fig. 1(c), (d), and (e) show *partial* assembly code of three procedures. Snippets (c) and (e) are taken from the `OpenSSL` procedure vulnerable to "Heartbleed", and were compiled using different compilers: `gcc` 4.9 and `CLang` 3.5, respectively. Snippet (d) is taken from an unrelated procedure in `Coreutils` and compiled using `gcc` 4.9. For simplicity and brevity, we only present a small part of the code.

Finding similarity between the procedures using syntactic techniques is challenging, as different compilers can produce significantly different assembly code. Instead, we decompose each procedure to small code segments we name *strands* (a strand is a basic-block slice, as explained in Section 3.2), and semantically compare the strands to uncover similarity, and lift the results into procedures.

In Fig. 1(c) & (e), the query code, q_2, and the target code, t, share three matching strands, numbered in the figure as ①, ②, and ③. Each strand is a sequence of instructions, and strands are considered as matches when they perform an equivalent computation. In the figure, we mark matching strands using the same circled number. Two syntactically different strands can be equivalent. For example, the strands numbered ② in q_2 and t differ syntactically but are equivalent (up to renaming and ignoring the change of `r9`). Furthermore, strands need not be syntactically contiguous. This is because they are based on data-flow dependencies rather than on syntactic properties. For example, strand ③ in the query procedure q_2 (`mov r12, rbx; lea rdi, [r12+3]`) matches the strand `mov r13, rbx;lea rcx, [r13+3]` in the target procedure t. In contrast, the code in Fig. 1(d) only matches the single strand ① in the target.

Our approach We present a novel notion of similarity for procedures that is based on the following key components:
Decomposing the procedure into strands: We decompose procedures into smaller segments we refer to as *strands*, which are feasible to compare.
Comparing strands: We use a program verifier [9] to check whether two strands are semantically equivalent by assuming input equivalence and checking if intermediate and output values are the same. When they are not equivalent, we define a quantitative notion of strand similarity based on the

proportion of matching values to the total number of values in the strand.

Statistical reasoning over strands: We present a statistical framework for reasoning about similarity of whole procedures using strand similarity. We compute a global similarity evidence score between procedures using the sum of the local evidence scores (*LES*) between strands. A key aspect of our framework is that we amplify the similarity scores of unique strands, expressed by a high *LES*, and diminish the significance of "common" strands (with respect to the target database examined) as they are less indicative of similarity.

Main contributions The contributions of this paper are:

- A framework for reasoning about similarity of procedures in stripped binaries. The main idea is to decompose procedures into strands, perform semantic comparison of them, and lift strand similarity into similarity between procedures using statistical techniques.
- A technique for checking input-output equivalence of strands of code, where all variables are unlabeled.
- A statistical model that quantifies the probability of similarity between procedures by examining the similarity of their strands.
- A prototype implementation in a tool called `Esh`, which is publicly available at github.com/tech-srl/esh. We compare `Esh` to previous binary code-search techniques using challenging search scenarios that combine patched and non-patched versions of real-world vulnerable procedures in binary form, compiled with different compiler versions and vendors. These experiments show that `Esh` achieves significantly better results.

```
lea     r14d, [r12+13h]
mov     r13, rax
mov     eax, r12d
lea     rcx, [r13+3]
shr     eax, 8
lea     rsi, [rbx+3]
mov     [r13+1], al
mov     [r13+2], r12b
mov     rdi, rcx
call    memcpy
mov     ecx, r14d
mov     esi, 18h
mov     eax, ecx
add     eax, esi
call    write_bytes
test    eax, eax
js      short loc_2A38
```

(a) `gcc v.4.9 -O3`

```
mov     r9, 13h
mov     r12, rax
mov     eax, ebx
add     rbp, 3
mov     rsi, rbp
lea     rdi, [r12+3]
mov     [r12+2], bl
lea     r13d, [rbx+r9]
shr     eax, 8
mov     [r12+1], al
call    _intel_memcpy
add     r9, 5h
mov     esi, r9d
mov     ecx, r13d
mov     eax, ecx
add     eax, esi
call    write_bytes
mov     ebx, eax
test    ebx, ebx
jl      short loc_342E
```

(b) `icc v.15.0.1 -O3`

Figure 2. Heartbleed vulnerability code snippets.

```
assume r12_q == rbx_t
```

```
v1_q = r12_q
v2_q = 13h + v1_q
v3_q = int_to_ptr(v2_q)
r14_q  = v3_q
v4_q = 18h
rsi_q  = v4_q
v5_q = v4_q + v3_q
rax_q  = v5_q
```

```
v1_t = 13h
r9_t = v1_t
v2_t = rbx_t
v3_t = v2_t + v1_t
v4_t = int_to_ptr(v3_t)
r13_t = v4_t
v5_t =  v1_t + 5
rsi_t = v5_t
v6_t = v5_t + v4_t
rax_t =  v6_t
```

```
assert v1_q == v2_t,v2_q == v3_t, v3_q == v4_t,r14_q == r13_t,
   v4_q == v5_t,rsi_q == rsi_t,v5_q == v6_t,rax_q == rax_t
```

Figure 3. Semantically similar strands.

2. Overview

In this section, we illustrate our approach informally using an example.

Given a query procedure from a stripped binary, our goal is to find similar procedures in other stripped binaries. To simplify presentation, we illustrate our approach on two code snippets instead of full procedures. Consider the assembly code of Fig. 2(a). This snippet is taken from a version of OpenSSL, which is vulnerable to the *Heartbleed* bug [3]. Our goal is to find similar vulnerable code snippets in our code-base. We would like to define a notion of similarity that can find matches even when the code has been modified, or compiled with a different compiler vendors and versions. For example, the code of Fig. 2(a) was compiled using `gcc v4.9`, and the code of Fig. 2(b), originating from the same source, was compiled using `icc v15.0.1`. We would like our approach to find the similarity of these two snippets despite their noticeable syntactic difference.

We focus on similarity rather than equivalence, as we would like our approach to apply to code that may have been patched. Towards that end, we compute a similarity score that captures similarity between (small) partial computations performed by each procedure.

The main idea of our approach is to decompose the code to smaller fragments, for which similarity is easy to compute, and use statistical reasoning over fragment similarity to establish the global similarity between code snippets. Towards that end, we have to answer three design questions:

- What is the best way to decompose the code snippets?
- How should the decomposed fragments be compared?
- How can fragment similarity be lifter into find snippet similarity?

Decomposition into strands We decompose a procedure into fragments which are feasible to compare. In this work, we use *strands*—partial dependence chains, as the basic unit. Fig. 3 shows two strands obtained from the code snippets of Fig. 2. For now, ignore the `assume` and `assert` operations added around the strands. The strands in the figure have been transformed to an Intermediate Verification Language (IVL), by employing tools from [5, 11]. The IVL abstracts away from specific assembly instructions, while maintaining the semantics of the assembly code. A fresh temporary variable is created for every intermediate value computed through-

```
v1_q = r14_q                    | v1_t = r14_t
v2_q = v1_q + 1                 | v2_t = v1_t + 16
v3_q = xor(v2_q,v1_q)           | v3_t = xor(v2_t,v1_t)
v4_q = and(v3_q,v2_q)           | v4_t = and(v3_t,v2_t)
v5_q = (v4_q < 0)               | v5_t = (v4_t < 0)
FLAGS[OF]_q = v5_q              | FLAGS[OF]_t = v5_t
```

Figure 4. Syntactically similar but semantically different strands.

out the execution. The flow of data between registers is always through these temporaries. Furthermore, the IVL always uses the full 64-bit representation of registers (e.g. `rax` and not `eax`) and represents operations on part of the register using temporaries and truncation (e.g. `mov rbx, al` will be `v1 = truncate(rax,8); rbx = v1;`). The strands in the figure have been aligned such that similar instructions of the two strands appear side by side. This alignment is only for the purpose of presentation, and our comparison is based on semantically comparing the strands. We added q and t postfixes to the strands' variables to separate the name spaces of variables in the two strands and specify one as the query and the other as the target. This allows us to create a joint program that combines the variables from the two strands and makes assumptions and assertions about their equality.

Comparing a pair of strands To compare a pair of strands, we create a joint program that combines them, but has a separate name space for the variables of each strand. We then explore the space of equality assumptions on the different inputs, and check how these assumptions affect the equality assertions on the outputs. For example, one choice for assumptions and assertions is shown in Fig. 3. Technically, given a pair of strands, we perform the following steps: (i) add equality assumptions over inputs of the two strands, (ii) add assertions that check the equality of all output variables (where output variables also include temporaries), and (iii) check the assertions using a program verifier and count how many variables are equivalent. Choosing which variables to pair when assuming and asserting equality is solved by searching the space of possible pairs. The choice of the strand as a small unit of comparison (with a relatively small number of variables), along with verifier based optimizations (described in Sec. 5.5), greatly reduce the search space, making the use of a verifier feasible.

Match Probability: We define an asymmetric similarity measure between a query strand, s_q, and a target strand, s_t, as the percentage of variables from s_q that have an equivalent counterpart in s_t. We denote this measure by $VCP(s_q, s_t)$. We later use this measure as a basis for computing the probability $Pr(s_q|s_t)$ that a strand s_q is input-output equivalent to a strand s_t (see Section 3.3.1).

For example, taking s_q to be the strand on the left-hand side of Fig. 3, and s_t to be the strand on the right-hand side, $VCP(s_q, s_t) = 1$, because all 8 variables from the left-hand side have an equivalent variable on the right-hand side. However, in the other direction, $VCP(s_t, s_q) = 8/9$. We note that no previous approach is able to produce such a matching, as the equivalent values are computed using different instructions.

In contrast to Fig. 3, the strands of Fig. 4 (taken from our corpus as specified in Section 5.1) are very similar syntactically (differ in just one character), but greatly differ semantically. Syntactic approaches would typically classify such pairs as matching, leading to a high rate of false positives. Our semantic comparison identifies that these two strands are different, despite their significant syntactic overlap, and yields $VCP = 1/6$, expressing the vast difference.

Local Evidence of Similarity: Modeling strand similarity using probability allows us to express other notions of similarity in a natural manner. For example, given a target procedure t and a query strand s_q, we can capture how well s_q can be matched in t by computing the maximal probability of $Pr(s_q|s_t)$ over any possible strand s_t in t.

We further define the probability, $Pr(s_q|H_0)$, of finding a matching strand for s_q at random (where H_0 represent all possible strands). The *significance* of finding a match for s_q in t can be then defined as:

$$LES(s_q|t) = \log \frac{\max_{s_t \in t} Pr(s_q|s_t)}{Pr(s_q|H_0)}.$$

LES provides a measure of the significance of the matching of s_q with t by comparing it to the matching of s_q with the random source H_0. It is important to measure the significance of a match, because many strand matches may be due to common strands introduced by the compiler (e.g., prolog/epilog), and therefore not significant in determining a match between procedures.

Lifting strand similarity into procedure similarity We say that two procedures are similar if one can be composed using (significantly similar) parts of the other. Given a query procedure q and a target procedure t, the $LES(s_q|t)$ measure indicates how significant the match of s_q is in t. We can therefore define global evidence of similarity (GES) between the procedures q and t by summing $LES(s_q|t)$ over all strands s_q in q (as further explained in Sec. 3.1).

The local and global similarity evidence allow us to lift semantic similarity computed between individual strands into a statistical notion of similarity between procedures.

Key Aspects of our approach

- Similarity by composition: we decompose procedures into strands, semantically compare strands, and lift strand similarity into procedure similarity using statistical techniques.
- Using strand similarity allows us to establish similarity between procedures even when the procedures are not equivalent, but still contain statistically significant similar strands.
- By performing semantic comparison between strands, we are able to find similarity across different compilers

versions and vendors (without knowledge of the specifics of the compilers). In Sec. 5 we compare our approach to previous work and show the importance of semantic comparison.

3. Strand-Based Similarity by Composition

In this section, we describe the technical details of our approach. In Sec. 3.1, we describe the steps for generating statistical procedure similarity. In Sec. 3.2 we illustrate how we decompose procedures into the single-path units of execution we call strands (step 1). We defer the exact details of computing similarity between strands (step 2) to Section 4, and assume that such a similarity measure is given. In Sec. 3.3 we define the likelihood and statistical significance of a strand, allowing us to quantify similarity using these probabilistic tools (step 3). In Sec. 3.4, we describe how whole procedure similarity is computed from statistical similarity of strands, using global and local evidence scores (step 4).

3.1 Similarity By Composition

We determine that two procedures are likely to be similar if non-trivial strands from one can be used to compose the other, allowing for some (compiler or patch related) transformation. We decompose the procedure to basic blocks, and then further split each block into strands using a slicing technique, as described in Sec. 3.2. We check for strand similarity with the aid of a program verifier, as further shown in Sec. 4. This gives us the flexibility to determine similarity between two strands, i.e., that they are (partially) input-output equivalent, even if they produce their result using different instructions, register allocation or program ordering. We then define a *Global Evidence Score*, $GES(q|t)$, representing the likelihood that a query procedure q is similar to a target procedure t. This global score is based on the composition of multiple *Local Evidence Scores* (denoted $LES(s_q|t)$) representing the sum of likelihoods that each strand $s_q \in q$ has a similar strand $s_t \in t$:

$$GES(q|t) = \sum_{s_q \in q} LES(s_q|t) = \sum_{s_q \in q} \log \frac{\max_{s_t \in t} Pr(s_q|s_t)}{Pr(s_q|H_0)}. \quad (1)$$

The right-hand side of Eq. 1 shows that the LES for strand s_q is calculated using the ratio between two factors: (i) the probability, $Pr(s_q|s_t)$, that s_q is semantically similar to one of the strands $s_t \in t$, where s_t is the strand that produces the highest probability (i.e., is most similar to s_q), and (ii) $Pr(s_q|H_0)$, the probability that this strand matches a random source. In our context, this means that the strand is commonly found in the corpus binaries and does not uniquely identify the query procedure. This is further described in Sec. 3.3.

3.2 Procedure Decomposition to Strands

We use a standard control flow graph (CFG) representation for procedures. We decompose the procedure by applying slicing [33] on the basic-block level. A *strand* is the set of instructions from a block that are required to compute a certain variable's value (backward slice from the variable). Each block is sliced until all variables are covered. As we handle each basic block separately, the inputs for a block are variables (registers and memory locations) used before they are defined in the block. Fig. 3 is an example of a pair of strands extracted from the blocks in Fig. 1.

Strands as partial program dependence graphs (PDGs) Strands are a practical compromise over enumerating all paths in the PDG [14]. All strands are obtained by decomposing the PDG at block boundaries. This means that strands only contain data dependencies, as control dependencies exist only over block boundaries, while severed data dependencies (e.g., values created outside the block) are marked as such and used in the comparison process (these are the inputs, as explained later in Algorithm 1). This approach of decomposing graphs while making use of loose edges to improve performance is similar to the "extended graphlets" in [17], yet there the edge contains less information as it is not connected to a certain variable. Decomposing at block level yielded precise results for most of our benchmarks. However, using longer paths can be advantageous when handling small procedures, where breaking at block level results in a small number of short blocks that can be easily matched with various targets. Sec. 6.6 further discusses benchmarks for which a the small number of blocks yields low accuracy for `Esh`.

Algorithm 1: Extract Strands from a Basic Block

```
Input: $b$ - An array of instructions for a basic-block
Output: $strands$ - $b$'s strands, along with their inputs
1  $unusedInsts \leftarrow \{1, 2, ..., |b|\}$; $strands \leftarrow []$;
2  while $unusedInsts \neq \emptyset$ do
3      $maxUsed \leftarrow \max(unusedInsts)$;
4      $unusedInsts \setminus= maxUsed$;
5      $newStrand \leftarrow [b[maxUsed]]$;
6      $varsRefed \leftarrow \texttt{Ref}(b[maxUsed])$;
7      $varsDefed \leftarrow \texttt{Def}(b[maxUsed])$;
8      for $i \leftarrow (maxUsed - 1)..0$ do
9          $needed \leftarrow \texttt{Def}(b[i]) \cap varsRefed$;
10         if $needed \neq \emptyset$ then
11             $newStrand$ += $b[i]$;
12             $varsRefed \cup= \texttt{Ref}(b[i])$;
13             $varsDefed \cup= needed$;
14             $unusedInsts \setminus= i$;
15     $inputs \leftarrow varsRefed \setminus varsDefed$;
16     $strands$ += $(newStrand, inputs)$;
```

Algorithm 1 uses standard machinery to extract strands from a basic block, and uses the standard notions of `Def` and `Ref` for the sets of variables defined and referenced (respectively) in a given instruction.

The process starts by putting all instructions in *unusedInsts*, and will end only when this list is empty, i.e., when every instruction is marked as having been used in at least one extracted strand. The creation of a new strand begins by taking the last non-used instruction, as well as initializing the list of variables referenced in the strand – *varsRefed*, and the variables defined in the strand – *varsDefed*. Next, all of the previous instructions in the basic-block are iterated *backwards*, adding any instruction which defines a variable referenced in the strand so far (is in *varsRefed*) and updating *varsRefed* and *varsDefed* with every instruction added. When the `for` loop is finished the new strand is complete, as every instruction needed to calculate all of the variables defined inside the basic-block is present. This does not include the *inputs*, which are any variables used in the calculation and not defined in the basic-block. Note that the backward iteration is crucial for minimizing the number of strands.

3.3 Statistical Evidence

Given two sets of strands obtained from two procedures for comparison, we assume a mechanism for computing a similarity measure between two strands based on the proportion of output variables they agree on when given equivalent inputs. We denote this similarity metric by VCP, and defer its formal definition and computation to Section 4. In this section, we assume that the VCP between two strands, s_q and s_t, is given, and transform it into a probabilistic measure for strand similarity $Pr(s_q|s_t)$. We then describe how to compute $Pr(s_q|H_0)$, the probability of each strand to match a random process. Using these values we compute the likelihood-ratio, $LR(s_q)$, from which the GES value is composed.

3.3.1 Strand Similarity as a Probability Measure

We denote $Pr(s_q|t)$ as the likelihood that a strand, s_q, from the query procedure, q, can be "found" in the target procedure t, i.e., that we can find an equivalent strand $s_t \in t$.

$$Pr(s_q|t) \triangleq \max_{s_t \in H_t} Pr(s_q|s_t). \qquad (2)$$

The likelihood $Pr(s_q|s_t)$ that two strands are input-output equivalent is estimated by applying a sigmoid function (denoted $g()$) over the VCP of the two strands (we set the sigmoid midpoint to be $x_0 = 0.5$ as $VCP(s_q, s_t) \in [0, 1]$):

$$Pr(s_q|s_t) \triangleq g(VCP(s_q, s_t)) = 1/(1 + e^{-k(VCP(s_q,s_t)-0.5)}). \qquad (3)$$

The use of the logistic function allows us to produce a probabilistic measure of similarity, where $Pr(s_q|t)$ is approximately 1 when $VCP(s_q, s_t) = 1$ and nears 0 when $VCP(s_q, s_t) = 0$. We experimented with different values to find the optimal value for the steepness of the sigmoid curve parameter, k, and found $k = 10.0$ to be a good value.

Our use of the sigmoid function is similar to its application in logistic regression algorithms for classification problems [18]. The hypothesis $h_\theta(x)$ is set to be the sigmoid function of the original hypothesis $\theta^T x$, resulting in the hypothesis being a probability distribution, $h_\theta(x) \triangleq Pr(y = 1|x; \theta) = g(\theta^T x)$, which reflects the likelihood of a positive classification ($y = 1$) given a sample x. This correlates to $Pr(s_q|s_t)$ representing the likelihood that s_q and s_t are a positive match for performing the same calculation.

3.3.2 The Statistical Significance of a Strand

In order to find procedure similarity in binaries, we require that non-trivial strands of code be matched across these binaries. This stands in contrast to Eq. 3, where smaller pieces of code will receive a high likelihood score, since they perform trivial functionality that can be matched with many strands. Thus a query strand need not only be similar to the target, but also have a *low probability to occur at random*. For this we introduce the *Likelihood Ratio* measure:

$$LR(s_q|t) = Pr(s_q|t)/Pr(s_q|H_0). \qquad (4)$$

This measure represents the ratio between the probability of finding a semantic equivalent of s_q in s_t vs. the probability of finding a semantic equivalent at random (from the random process H_0). $Pr(s_q|H_0)$ in fact measures the statistical insignificance of a strand, where a higher probability means low significance. We estimate the random hypothesis H_0 by averaging the value of $Pr(s_q|s_t)$ over all targets (as it needs to be computed either way), i.e., $Pr(s_q|H_0) = \frac{\sum_{s_t \in T} Pr(s_q|s_t)}{|T|}$ where T is the set of all target strands for all targets in the corpus.

3.4 Local and Global Evidence Scores

After presenting the decomposition of procedures q and t to strands, and defining the likelihood-ratio for each strand s_q and target t, we can define the *Local Evidence Score* as the log of the likelihood-ratio:

$$LES(s_q|t) = \log LR(s_q|t) = \log Pr(s_q|t) - \log Pr(s_q|H_0). \qquad (5)$$

This local score reflects the level of confidence for s_q to have a non-trivial semantic equivalent strand in t. The global score $GES(q|t)$ is simply a summation (Eq. 1) of all $LES(s_q|t)$ values after decomposing q to strands, which reflects the level of confidence that q can be composed from non-trivial parts from t and is in fact similar to it.

4. Semantic Strand Similarity

In the previous section, we assumed a procedure that computes semantic strand similarity. In this section, we provide such a procedure using a program verifier. Given two strands to compare, the challenge is to define a quantitative measure of similarity when the strands are not equivalent. We first provide the formal definitions on which the semantics of strand similarity are based, and then show how we compute strand similarity using a program verifier.

4.1 Similarity Semantics

Preliminaries We use standard semantics definitions: A *program state* σ is a pair $(l, values)$, mapping the set of program variables to their concrete value $values : Var \to Val$, at a certain program location $l \in Loc$. The set of all possible states of a program P is denoted by Σ_P. A *program trace* $\pi \in \Sigma_P^*$ is a sequence of states $\langle \sigma_0, ..., \sigma_n \rangle$ describing a single execution of the program. The set of all possible traces for a program is denoted by $[\![P]\!]$. We also define $first : \Sigma_P^* \to \Sigma_P$ and $last : \Sigma_P^* \to \Sigma_P$, which return the first and last state in a trace respectively.
A *strand* $s \in P$ is therefore a set of traces, $s \subseteq [\![P]\!]$ – the set of all traces generated by all possible runs of s, considering all possible assignments to s inputs. We will use this abstraction to further define strand equivalence and VCP.

Variable correspondence A variable correspondence between two states, σ_1 and σ_2, denoted $\gamma : Var_1 \nrightarrow Var_2$, is a (partial) function from the variables in σ_1 to the variables in σ_2. Note that several variables can be mapped to a single variable in Var_2. $\Gamma(P_1, P_2)$ denotes the set of all variable correspondences for the pair of programs (P_1, P_2). This matching marks the variables as candidates for input-output equivalence to be proven by the verifier.

State, trace equivalence Given two states and a correspondence γ, if $\forall (v_1, v_2) \in \gamma : \sigma_1(v_1) = \sigma_2(v_2)$, then we say that these states are equivalent with respect to γ, and denote them $\sigma_1 \equiv_\gamma \sigma_2$. Given two traces and a correspondence γ between their *last states*, if $last(\pi_1) \equiv_\gamma last(\pi_2)$, then we say that these traces are equivalent with respect to γ, and denote them $\pi_1 \equiv_\gamma \pi_2$.

Definition 1 (*Strand equivalence*)**.** Given two strands (remembering that each strand has inputs as defined in Sec. 3.2 and denoted $inputs(s)$) and a correspondence γ, we say that these strands are equivalent with respect to γ, denoted $s_1 \equiv_\gamma s_2$ if: (i) *every input* from s_1 is matched with some input from s_2 under γ, and (ii) every pair of traces $(\pi_1, \pi_2) \in (s_1, s_2)$ that agree on inputs $(\forall (i_1, i_2) \in (\gamma \cap (inputs(s_1) \times inputs(s_2))) : first(\pi_1)(i_1) = first(\pi_2)(i_2))$ is equivalent $\pi_1 \equiv_\gamma \pi_2$. This expresses input-output equivalence.

Definition 2 (*State, trace variable containment proportion*)**.** We define the VCP between a query state σ_q and a target state σ_t as the *proportion* of matched values in σ_q, denoted $VCP(\sigma_q, \sigma_t) \triangleq \frac{|\gamma_{max}|}{|\sigma_q|}$, where γ_{max} is the maximal variable correspondence (in size) for which the two states are equivalent, i.e., $\sigma_q \equiv_{\gamma_{max}} \sigma_t$, considering all possible gammas. We define the VCP between two traces, $VCP(\pi_q, \pi_t)$, as $VCP(last(\pi_q), last(\pi_t))$.

For instance, given $values_q = \{x \mapsto 3, y \mapsto 4\}$, $values_t = \{a \mapsto 4\}$, the maximal correspondence is therefore $\gamma_{max} = \{y \mapsto a\}$ as it matches the most possible variables. Therefore $VCP(\sigma_q, \sigma_t) = \frac{1}{2}$. We note that it is possible for several maximal correspondences to exist, and in these cases we simply pick one of the said candidates.

Definition 3 (*Strand VCP*)**.** We define the VCP between two strands as the proportion of matched variables in the γ that induces the maximal containment proportion over *all pairs* of traces, as follows:

$$VCP(s_q, s_t) \triangleq \frac{\max\{|\gamma| \big| \forall (\pi_q, \pi_t) \in (s_q, s_t) : \pi_q \equiv_\gamma \pi_t\}}{|Var(s_q)|}.$$

An important observation regarding the VCP is that it can produce a high matching score for potentially unrelated pieces of code, for instance if two strands perform the same calculation but one ends by assigning 0 to all outputs, or if the result of the computation is used for different purposes. We did not observe this to be the case, as (i) compiler optimizations will eliminate such cases where a computation is not used, and (ii) even if the code is used for different purposes – it may still suggest similarity, if for example a portion of the query procedure was embedded in the target.

4.2 Encoding Similarity as a Program Verifier Query

Next we show how we compute a strand's VCP (Def. 3) by encoding input-output equivalence, along with procedure semantics as a program verifier query. The query consists of three parts, including (i) assuming input equivalence over the inputs in the variable correspondence (γ), (ii) expressing query and target strand semantics by sequentially composing their instructions, and (iii) checking for variable equivalence, over all possible traces, by adding equality assertions to be checked by the program verifier.

Program verifiers For the purposes of this paper, we describe a *program verifier* simply as a function denoted *Solve* : (*Proc,Assertion*) $\to$ (*Assertion* $\to$ {*True, False*}), that given a procedure $p \in$ *Proc* with inputs $i_1, ...i_n$ and a set of assertion statements $\Phi \subseteq$ *Assertion*, is able to determine which of the assertions in Φ hold, for *any* execution of p, under all possible values for $i_1, ..., i_n$. The assertions in Φ are mapped to a specific location in p and specify a property (a formula in first-order logic (FOL)) over p's variables that evaluates to *True* or *False* according to variable value. *Solve* will label an assertion as *True* if for all variable values under all input values, the assertion holds. Verifiers usually extend the program syntax with an `assume` statement, which allows the user to specify a formula at desired program locations. The purpose of this formula is to instruct the verifier to assume the formula to always be true at the location, and try to prove the assertions encountered using all the assumptions encountered in the verification pass. In this work, we used the `Boogie` program verifier. ([9] provides the inner workings of the verifier.) To allow the use of the verifier, we lifted assembly code into a non-branching subset of the (`C`-like) `Boogie` IVL (translation details described in Sec. 5.1.1). The details of `BoogieIVL` are throughly described in [21].

Procedure calls We treat procedure calls as uninterpreted functions while computing similarity because, (i) an interprocedural approach would considerably limit scalability, as the verifier would need to reason over the entire call tree of the procedure (this could be unbounded for recursive calls), and (ii) we observed that the semantics of calling a procedure is sufficiently captured in the code leading up to the call where arguments are prepared, and the trailing code where the return value is used. Calls to two different procedures that have the exact same argument preparation process and use return values in the same way would be deemed similar by our approach. We note, however, that our approach does not rely on knowing call targets, as most are omitted in stripped binaries.

Algorithm 2: Compute Strand VCP

```
Input: Query $p^q$, Target $p^t$ in BoogieIVL
Output: $VCP(p^q, p^t)$
1  $maxVCP \leftarrow 0$;
2  for $\gamma \in \Gamma(p^q, p^t)$ do
3      $p \leftarrow$ NewProcedure $(Inputs(p^q) \cup Inputs(p^t))$;
4      for $(i^q, i^t) \in (\gamma \cap (Inputs(p^q) \times Inputs(p^t))$ do
5          $p$.body.Append (assume $i^q == i^t$);
6      $p$.body.Append ($p^q$.body;$p^t$.body);
7      for $(v^q, v^t) \in (((Vars(p^q) \times Vars(p^t)) \cap \gamma)$ do
8          $p$.body.Append (assert $v^q == v^t$);
9      Solve ($p$);
10     if $p^q \equiv_\gamma p^t$ then
11         $maxVCP \leftarrow \max(|\gamma|/|Vars(p^q)|, maxVCP)$;
```

Calculating strand VCP ***using a program verifier*** Next, we describe how we encode strand similarity as a Boogie procedure. We present a simplified version of the algorithm for clarity and brevity, and further describe optimizations in Sec. 5.5. As our compositional approach alleviates the need to reason over branches, we can define the encoding assuming single-path programs. For the rest of the paper we separate procedure variables to $Vars(p)$, denoting all non-input variables in the procedure, and $Inputs(p)$, denoting only inputs. Algorithm 2 receives a pair of Boogie procedures p^q, p^t representing the strands q and t, after renaming of variables to avoid naming collisions. It then proceeds to enumerate over all possible variable correspondences $\gamma \in \Gamma(p^q, p^t)$, where all of p_q's inputs are matched in compliance with Def. 1. For each correspondence, a new Boogie procedure p is created. We start building the procedure body by adding assumptions of equivalence for every pair of inputs in γ. This is crucial for checking input-output equivalence. Next, we append the bodies of the query and target procedures sequentially, capturing both strands' semantics. Lastly, a series of assertion statements are added, whose goal is to assert the exit state equivalence by adding an assertion for all variable pairs matched by γ. The resulting procedure p is then given to the Solve() function, which uses the program verifier to check assertion correctness. If all the assertions were proven, the current VCP is calculated and compared against the best VCP computed so far, denoted $maxVCP$. The higher value is picked, resulting in the maximal VCP at the end of the loop's run.

5. Evaluation

Our method's main advantage is its ability to perform cross-compiler (and cross-compiler-version) code search on binary methods, even if the original source-code was slightly patched. We will harness this ability to find vulnerable code; we demonstrate our approach by using a known (to be) vulnerable binary procedure as a query and trying to locate other similar procedures in our target database that differ only in that they are a result of compiling slightly patched code or using a different compilation tools. These similar procedures become suspects for being vulnerable as well and will be checked more thoroughly. We designed our experiments to carefully test every major aspect of the problem at hand. Finally, we will also show how our method measures up against other prominent methods in this field.

In Sec. 5.1 we describe the details of our prototype's implementation. In Sec. 5.2 we detail the vulnerable and non-vulnerable code packages that compose our test-bed. In Sec. 5.3 we show how our test-bed was built and how this structure enabled us to isolate the different aspects of the problem domain. In Sec. 5.4 we explain the ROC & CROC classifier evaluation tools that we used to evaluate our method's success. In Sec. 5.5 we detail the different heuristics we employed in our prototype to improve performance without undermining our results.

5.1 Test-Bed Creation and Prototype Implementation

We implemented a prototype of our approach in a tool called Esh. Our prototype accepts a query procedure and database of procedures (the targets), residing in executable files, as input in binary form. Before the procedures can be compared using our method, we need to divide them (from whole executables into single procedures) and "lift" them into the BoogieIVL to enable the use of its verifier.

5.1.1 Lifting Assembly Code into BoogieIVL

Each binary executable was first divided to procedures using a custom IDA Pro (the Interactive DisAssembler) [4] Python script, which outputs a single file for every procedure. BAP (Binary Analysis Framework) [11] "lifts" the binary procedure into LLVM IR [20] code, which manipulates a machine state represented by global variables. An important observation, and this was shown in Fig. 3 & Fig. 4, is that translated code is in Single Static Assignment (SSA) form, which is crucial for an effective calculation of the VCP (Def. 3). The SMACK (Bounded Software Verifier) [5] translator is used to translate the LLVM IR into BoogieIVL. Finally, strands are extracted from the blocks of the procedure's CFG.

Alongside Esh, we performed all the experiments on the prototype implementation of [12] called TRACY. Sec. 6.3 shows a detailed analysis of these experiments.

Our prototype was implemented with a mixture of C# (for Boogie framework interaction) Python. The full source code & installation guide are published on Github, and the prototype was deployed on a server with four Intel Xeon E5-2670(2.60GHz) processors, 377 GiB of RAM, running Ubuntu 14.04.2 LTS.

5.2 Using Vulnerable Code as Queries

To make sure the experiments put our method to a real-world test scenario, we incorporated eight real vulnerable code packages in the test-bed. The specific Common Vulnerabilities and Exposures (CVEs) are detailed in Tab. 1. The rest of the target database was composed from randomly selected open-source packages from the Coreutils [2] package.

All code packages were compiled using the default settings, resulting in most of them being optimized using the -O2 optimization level while a few, like OpenSSL, default to -O3. All executables were complied to the x86_64 (64-bit) architecture as default. Following this we decided to focus our system's implementation efforts on this architecture. Note that our system can be easily expanded to support x86 (32bit) and even other chip-sets (assuming our tool-chain provides or is extended to provide support for it). After compilation we removed all debug information from the executables. After compiling the source-code into binaries, our target corpus contained 1500 different procedures.

5.3 Testing Different Aspects of the Problem Separately

Many previous techniques suffered from a high rate of false positives, especially as the code corpus grew. We will show one cause for such incidents using a general example.

[28] shows that the compiler produces a large amount of compiler-specific code, such as the code for its control structures, so much so that the compiler can be *identified* using this code. This is an example of a common pitfall for methods in the fields of binary code search: if the comparison process is not precise enough when comparing procedures compiled by a different compiler, a low similarity score can be wrongly assigned for these procedures on the basis of the generating compiler alone. This is due to the compiler-centric code taking precedence over the real semantics of the code, instead of being identified as common code and having its importance reduced in the score calculation.

In our context, the problem can be divided into three vectors: (i) different compiler versions, (ii) different compiler vendors, and (iii) source-code patches.

Different versions of the same compiler Before attempting cross-compiler matching, we first evaluated our method on binaries compiled using different versions of the same compiler. We therefore compiled the last vulnerable version of each package mentioned in Sec. 5.2 using the gcc compiler versions 4.{6,8,9}. Next we performed the same process with the CLang compiler versions 3.{4,5}, and again with the icc compiler versions 15.0.1 and 14.0.4.

Cross-compiler search Next, we evaluated our approach by searching procedures compiled across different compilers. An important aspect of this evaluation process was alternating the query used, each time selecting the query from a different compiler. This also ensured that our method is not biased towards a certain compiler. As explained in Section 3, our matching method is not symmetric, and so examining these different scenarios will provide evidence for the validity of this asymmetric approach.

Patched source-code The last vector we wish to explore is patching. We define a patch as any modification of source-code that changes the semantics of the procedure. The common case for this is when the procedure's code is altered, yet changes to other procedures or data-structures can affect the semantics as well. We predict that precision will decline as the size of the patch grows and the procedures exhibit greater semantic difference.

5.4 Evaluating Our Method

A naive approach to evaluating a method which produces a quantitative similarity score is to try and find a "noise threshold". This threshold transforms the quantitative method into a binary classifier by marking all pairs of procedures with a score above the threshold as a match, and the rest as a non match.

Alas, for most cases there is no clear way to compute or detect one threshold which creates a clear separation between true and false positives for *all* experiments. As this is also true for our method, we will evaluate our tool by examining the results of our experiments as a ranked list, and use a measure which reflects whether the true positives are ranked at the top of that list.

Evaluating classifiers with ROC The receiver operating characteristic (ROC) is a standard tool in evaluation of threshold based classifiers. The classifier is scored by testing all of the possible thresholds consecutively, enabling us to treat each method as a binary classifier (producing 1 if the similarity score is above the threshold). For binary classifiers, accuracy is determined using the True Positive (TP, the samples we know are positive), True Negative (TN, the samples we know are negative), Positive (P, the samples classified as positive) and Negative (N, the samples classified as negative) as follows: $Accuracy = (TP + TN)/(P + N)$. Plotting the results for all the different thresholds on the same graph yields a curve; the area under this curve (AUC) is regarded as the accuracy of the proposed classifier.

Evaluating classifiers with CROC Concentrated ROC [32] is an improvement over ROC that addresses the problem of "early retrieval" – where the corpus size is huge and the number of true positives is low. The idea behind the

CROC method is to better measure accuracy in a scenario with a low number of TPs. The method assigns a higher grade to classifiers that provide a low number of candidate matches for a query (i.e., it penalizes false positives more aggressively than ROC). This is appropriate in our setting, as *manually verifying a match is a costly operation for a human expert*. Moreover, software development is inherently based on re-use, so similar procedures should not appear in the same executable (so each executable will contain at most one TP).

5.5 Enabling the Use of Powerful Semantic Tools

Esh Performance Our initial experiments showed the naive use of program verifiers to be infeasible, resulting in many hours of computation for each pair of procedures. In the following subsection we describe various optimizations to our algorithm which reduced the time required for comparing a pair of procedures to roughly 3 minutes on average on an 8-core `Ubuntu` machine. We emphasize that our approach is *embarrassingly parallelizable*, as verifier queries can be performed independently, allowing performance improvements linearly to the number of computation cores.

Algorithm 2 optimizations We first presented the VCP computation algorithm in a simplified form with no optimizations. To avoid enumerating over all variable correspondences in $\Gamma(p^q, p^t)$, in our optimized implementation we enumerated over inputs only $(Inputs(p^q) \times Inputs(p^t))$. Furthermore, we did not allow multiple inputs in p^q to be matched with a single input in p^t (γ was one-to-one) and only allowed for correspondences that matched all of p^q inputs. This reduced the number of outer loop iterations to $\max(|I_q|!, |I_t|!)$. We further reduced this by maintaining *typing* in matches.

For each matching of the inputs, the non-input variable matching part of γ starts out simply as $Var(p^q) \times Var(p^t)$, while maintaining types. We further perform a data-flow analysis to remove variable pairs that have no chance of being matched, as their calculation uses inputs that were not matched with an initial assumption (were not in γ). Allowing for all possible matchings (up to typing and dataflow analysis) means that we check *all possible correspondences for non-inputs at once*. We do this by parsing the output of the `Boogie` verifier that specifies which of the equality assertions hold and which fail. Unmatched variables are removed, leaving only equivalent pairs in γ. Finally, multiple matchings for variables are removed (γ must be a function over q's variables according to Def. 3) and the VCP is calculated.

Discarding trivial strands and impractical matches As mentioned in Sec. 3.3, trivial strands receive low LES similarity scores. Thus smaller strands are less likely to generate evidence for global similarity GES. We therefore established a minimal threshold for the number of variables required in a strand in order that it be considered in the similarity computation (5 in our experiments), and did not produce verifier queries for strands smaller than the said threshold. We further avoided trying to match pairs of strands which vary greatly in size. We used a ratio threshold, set to 0.5 in our experiments, meaning that we will attempt to match a query strand only with target strands that have at least half the number of variables (i.e. a minimal value of $VCP = 0.5$ is required) or at most twice the number of variables (avoiding matching with "giant" strands, which are likely to be matched with many strands).

Batching verifier queries To further save on the cost of performing a separate verifier query for each input matching (which usually contains a small number of assertions), we batch together multiple queries. As we wanted to allow procedure knowledge gathered by the verifier to be reused by subsequent queries, we embedded several possible γ correspondences in the same procedure by using the non-deterministic branch mechanism in `Boogie` (further explained in [21]) to make the verifier consider assumptions and check assertions over several paths. We also batched together different strand procedures up to a threshold of 50,000 assertions per query.

6. Results

6.1 Finding `Heartbleed`

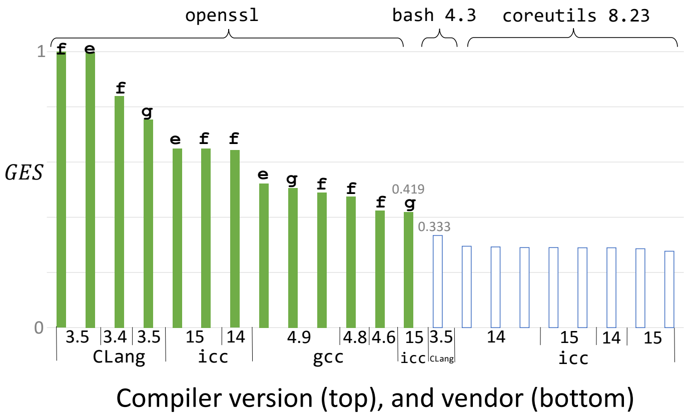


Figure 5. Experiment #1 – Successfully finding 12 procedure variants of `Heartbleed`.

To better understand our evaluation process and test-bed design, we start by walking through experiment #1 from Tab. 1. The query for this experiment was the "Heartbleed" vulnerable procedure from `openssl-1.0.1f`, compiled with `CLang 3.5`. Each bar in Fig. 5 represents a single target procedure, and the height of the bar represents the GES similarity score (normalized) against the query. The specific compiler vendor and version were noted below the graph (on the X axis) the source package and version above it. Bars filled green represent procedures originating from the same code as the query (i.e. "Heartbleed") but vary in compilation or source code version (the exact code version `openssl-1.0.1{e,g,f}` is

#	Alias/Method				S-VCP			S-LOG			Esh		
		CVE/Stats	#BB	# Strands	FP	ROC	CROC	FP	ROC	CROC	FP	ROC	CROC
1	Heartbleed	2014-0160	15	92	107	0.967	0.814	0	1.000	1.000	0	1.000	1.000
2	Shellshock	2014-6271	136	430	246	0.866	0.452	3	0.999	0.995	3	0.999	0.996
3	Venom	2015-3456	13	67	0	1.000	1.000	0	1.000	1.000	0	1.000	1.000
4	Clobberin' Time	2014-9295	41	233	351	0.797	0.343	65	0.987	0.924	19	0.993	0.956
5	Shellshock #2	2014-7169	88	294	175	0.889	0.541	40	0.987	0.920	0	1.000	1.000
6	ws-snmp	2011-0444	6	98	42	0.981	0.879	5	0.999	0.990	1	1.000	0.997
7	wget	2014-4877	94	181	332	0.885	0.600	11	0.998	0.988	0	1.000	1.000
8	ffmpeg	2015-6826	11	87	222	0.9212	0.6589	97	0.9808	0.8954	0	1.000	1.000

Table 1. The ROC, CROC and false positives (FP) for our query search experiments

specified over the bar itself). All unrelated procedures were left blank.

As we can see from the results in Fig. 5, our method gives high scores to *all other similar versions* of the "Heartbleed" procedure, despite them being compiled using different compilers, different compiler versions or from a patched source code. A gap of 0.08 in the GES score exists between the true positives from the rest of the procedures. (0.419 for the `icc 15` compiled procedure of `openssl-1.0.1g` "Heartbleed" vs. 0.333 for the `bash 4.3` "ShellShock" procedure compiled with `Clang 3.5`). It is important to note that we will not try to establish a fixed threshold to evaluate the quality of these results. As mentioned, this clean separation between the true positives and the false positives is not always possible. Instead, this result and others, as shown in the following sections, are evaluated according to the produced ranking. The result in Fig. 5 receives a $ROC = CROC = 1.0$ score as it puts all of the true positives in the top of the ranking.

6.2 Decomposing Our Method into Sub-methods

When examining our method bottom up, we can divide it into three layers:

- S-VCP: The first layer of our method is the way we calculate the VCP between strands. Without the use of the statistical processing, we still define a similarity score as: $\sum_{s_t \in T} \max_{s_q \in Q}(VCP(s_t, s_q))$. This approach attempts to generalize the VCP from a pair of strands to a pair of procedures by counting the maximal number of matched variables in the *entire* procedure.
- S-LOG: The next layer of our approach incorporates the statistical significance of every query strand, by using local and global significance. By alternatively defining $Pr(s_q, s_t) = VCP(s_q, s_t)$ and applying it to the LES and GES equations, we can see how our method looks without applying the sigmoid function to the VCP.
- `Esh`: Adding the use of the sigmoid function results in our full method as described in Section 3

To fully justify and explain each layer of our method, Tab. 1 shows the results of each sub-method compared with our full method, in terms of (i) false positives (FT), and (i) ROC & CROC (explained in Sec. 5.4). Note that we count the number of false positives as determined by a human examiner who receives the list of procedures sorted by similarity scores, and we define the number of false positives as the number of non-matching procedures the human examiner will have to test until all the true similar procedures are found. The effectiveness of a method can be measured more precisely and quickly by using CROC. We also included additional information about the number of basic-blocks and the number of strands extracted from them, and the CVE for every vulnerable procedure we attempted to find.

These results clearly show that each layer of our method increases its accuracy. Comparison of the different experiments shows that CROC & ROC scores do more than simply count the number of false positives for every threshold they compare the *rate* by which the false positive rate *grows*. (Informally, this may be regarded as a prediction of the number of attempts after which the human researcher will give up.) An important point is that the size of the query, in terms of the number of basic blocks or strands, does not directly correlate with easier matching.

Experiment #3 shows an interesting event, where even S-VCP gets a perfect score. Upon examination of the query procedure, we discovered that this occurs because the said procedure contains several distinct numeric values which are only matched against similar procedures. (These are used to explore a data structure used to communicate with the `QEMU` floppy device.)

Examining the results as a whole, we see that in more than half of the experiments the use of S-VCP, which doesn't employ the statistical amplification of strands, results in a high number of false positives. To understand this better, we performed a thorough analysis of experiment #5. When we examine the values of $Pr(s_q|H_0)$, which express the frequency of appearance of strand s_q, we see that several strands get an unusually high score (appear more frequently). One of these strands was found to be a sequence of `push REG` instructions, which are commonplace for a procedure prologue.

6.3 Comparison of `TRACY` and `Esh`

Following our explanation of the different problem aspects in Sec. 5.3, we tested both tools, with a focus on experiment number one in Tab. 1. Each line in Tab. 2 represents a single experiment; the ✓in one of the columns specifies that this specific problem aspect is applied: (i) Compiler version from the same vendor, (ii) Cross-compiler, meaning

Versions	Cross	Patches	TRACY (Ratio-70)	Esh
✓			1.0000	1.0000
	✓		0.6764	1.0000
		✓	1.0000	1.0000
✓	✓		0.5147	1.0000
	✓	✓	0.4117	1.0000
✓		✓	0.8230	1.0000
✓	✓	✓	0.3529	1.0000

Table 2. Comparing TRACY and Esh on different problem aspects.

different compiler vendors, and (iii) applying patches to the code. For example, when (i) and (ii) are checked together this means that all variations of queries created using all compiler vendors and compiler versions were put in the target database.

As we can see, because TRACY was designed to handle patches, it achieves a perfect grade when dealing them. Moreover, TRACY can successfully handle the different compiler versions *on their own*. However when it is used in a cross-compiler search, its accuracy begins to plummet. Furthermore, when any two problem aspects are combined, and especially when all three are addressed, the method become practically unusable.

6.4 Evaluating BinDiff

BinDiff [6] is a tool for comparing whole executables/libraries by matching all the procedures within these libraries. It works by performing syntactic and structural matching relying mostly on heuristics. The heuristic features over procedures which are the basis for similarity include: the number of jumps, the place of a given procedure in a call-chain, the number of basic blocks, and the name of the procedure (which is unavailable in stripped binaries). Detailed information about how this tool works can be found in [7], along with a clear statment that BinDiff ignores the semantics of concrete assembly-level instructions.

Tab. 3 shows the results of running BinDiff on each of the procedures in Tab. 1. As BinDiff operates on whole executables/libraries, the query/target was a whole library containing the original vulnerability. We only compared one target for each query, the same library compiled with a different vendor's compiler, and patched (for the queries where patching was evaluated). BinDiff failed to find the correct match in all experiments but two. The experiments in which BinDiff found the correct match are those where the number of blocks and branches remained the same, and is relatively small, which is consistent with [7].

6.5 Pairwise Comparison

Fig. 6 shows the similarity measures produced in an all-vs-all experiment, where 40 queries were chosen at random from our corpus and compared. The result is shown in heat-map form, where the axes represent individual queries (X and Y are the same list of queries, in the same order) and each pixel's intensity represents the similarity score *GES*

Alias	Matched?	Similarity	Confidence
Heartbleed	✗	-	-
Shellshock	✗	-	-
Venom	✗	-	-
Clobberin' Time	✗	-	-
Shellshock #2	✗	-	-
ws-snmp	✓	0.89	0.91
wget	✗	-	-
ffmpeg	✓	0.72	0.79

Table 3. Evaluating BinDiff

value (Eq. 1) for the query-target pair. Queries that originate from the same procedure (but are compiled with different compilers, or patched) are coalesced. Different procedures are separated by ticks. We included at least two different compilations for each procedure. The first procedure (left-most on the X, bottom on Y axis) is ftp_syst() from wget 1.8, queried in 6 different compilations. The second is taken from ffmpeg 2.4.6, queried in 7 different compilations. The rest are taken from Coreutils 8.23. The average *ROC* and *CROC* values for the experiment were 0.986 and 0.959 respectively.

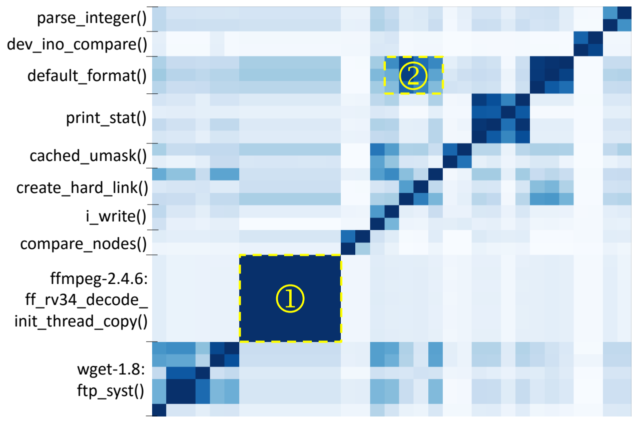


Figure 6. All vs. all experiment result in heat map form (pixel intensity = similarity *GES* measure).

Several observations can be made from Fig. 6:

1. The graph's diagonal represents the "ground truth" i.e., each query is matched perfectly with itself.
2. Our *GES* measure is not symmetrical (as it is based on the asymmetrical *VCP* metric).
3. Esh provides a very clear distinction for ffmpeg-2.4.6's ff_rv34_decode_init_thread_copy(), marked with a dashed region numbered ①, where all compiled queries of the procedure receive high *GES* scores when compared with each other and low ones when compared with random code. In general, Esh correctly matches procedures compiled with different compilers, as the pattern of "boxes" along the diagonal shows.

4. Groups of queries that originate from the same procedure generally receive similar *GES* values (i.e. similar shade) when compared to groups of targets that originate from the same (but different from the queries') code.

Although some procedures, such as `default_format()`, seem to perform poorly as they have high *GES* matches with wrong targets (marked with a dashed region numbered ②), the correct evaluation of the matching should be relative to the correct matching. For `default_format()`, the *GES* values of the correct matching is high (specified by dark pixels around the diagonal), so relative to that, the set of wrong matchings (somewhat dark pixels in the middle) becomes less significant, which is reflected by a $ROC = .993$ and $CROC = .960$ AUC score.

6.6 Limitations

In this subsection we discuss the limitations of our approach using study cases of procedures where `Esh` yields low recall and precision.

"Trivial" procedures and Wrappers Our approach relies on matching semantically similar, non-trivial fragments of execution. Thus, when matching a very small query (i.e. a "trivial" procedure), we are forced to rely on the statistical significance of a very small number of relatively short strands, which yields poor results. Wrappers are procedures that mostly contain calls to other procedures, and hold very little logic of their own e.g. the `exit_cleanup()` procedure in Fig. 7. As we operate on stripped binaries, we cannot rely on identifiers like procedure names to try and match strands based on procedure calls.

```
static void exit_cleanup (void) {
  if (temphead)
    {
      struct cs_status cs = cs_enter ();
      cleanup ();
      cs_leave (cs);
    }
  close_stdout ();
}
```

Figure 7. Wrapper `exit_cleanup` procedure from `sort.c` of `Coreutils` 8.23

Generic procedures The `C` concatenation preprocessor directive (`##`) is sometimes used as a mechanism for generics, where "template" procedures are created, that have similar structure but vary in type, or use a function pointer to provide different functionality as shown in Fig. 8, where different string comparison procedures are created to compare creation, modification and access time of a file. As these hold the same general structure (and are specifically small), `Esh` deems the whole set of procedures similar. We do not consider these matches to be strict false positives, as the procedures are in fact similar in structure. Note that this problem only arises when these types of procedures are used as queries, as it is not clear whether the different variants should be considered as true positives (they perform similar semantics) or false positives (as they might operate on different data structures).

```
#define DEFINE_SORT_FUNCTIONS(key_name, key_cmp_func) \
  static int strcmp_##key_name (V a, V b) \
  { return key_cmp_func (a, b, strcmp); }

  ...

DEFINE_SORT_FUNCTIONS (ctime, cmp_ctime)
DEFINE_SORT_FUNCTIONS (mtime, cmp_mtime)
DEFINE_SORT_FUNCTIONS (atime, cmp_atime)
```

Figure 8. `DEFINE_SORT_FUNCTIONS` macro for creating "template" procedures in `ls.c` of `Coreutils` 8.23

7. Related Work

In this section, we briefly describe closely related work.

Equivalence checking and semantic differencing [23, 24, 27] are aimed at proving equivalence and describing differences between (versions of) procedures in high-level code, and do not apply to searching for similarity in machine code. The authors of [23, 24] offer some handling for loops, but apply computationally expensive abstract domain libraries, which do not scale in our setting. Sharma et al. [30] present a technique for proving equivalence between high-level code and machine code, with loops, by trying to find a simulation relation between the two. However the search is computationally demanding and will not scale. Lahiri et al. [19] narrow the scope to assumed-to-be but cannot-be-proved-to-be-equivalent snippets of binary code, and attempts to find a set of changes (add or delete) to complete the proof. While having interesting implications for our problem, the assumption of a variable mapping encumbers the transition. Ng and Prakash [22] use symbolic execution for similarity score calculation, but it is not geared towards cross-compiler search, and is limited to handling patched procedures (specifically, only handles one calling convention and rejects procedures based on the number of inputs.)

Compiler bug-finding Hawblitzel et al. [15] present a technique that handles compiled versions of the same procedure from different compilers. Their goal is to identify root causes of compiler bugs, and their approach cannot be directly applied to our setting as they: (i) require strict equivalence and thus even a slight change would be deemed a bug, (ii) know the IL originated from the same code allowing them to easily match inputs and outputs (i.e. these are *labeled*) for solving, which is not the case for machine code, and (iii) offer no method for segmenting procedures and thus are limited in handling loops (they use loop unrolling up to 2 iterations).

Dynamic methods Egele et al. [13] present a dynamic approach which executes the procedure in a randomized environment and compares the side effects of the machine for similarity. As they base the similarity on a single randomized run, similarity may occur by chance, especially since they coerce execution of certain paths to achieve full coverage. Their evaluation indeed outperforms [6]. However, it is still wanting as they rank similar functions in the top 10 in only 77% of the cases, and do not evaluate over patched versions.

An interesting method by Pewny et al. [25] uses a transition to an intermediate language (VEX-IR), a simplification using the Z3 theorem prover, sampling and a hashing-based comparison metric. In their results they report several problems with false positives. We believe that this is because sampling alone is used to match two basic-blocks without proving the match, and that basic-blocks are not weighted against how common they are in the corpus (and these basic-blocks might be a compilation by-product). Moreover, the goal of their method is to find clones for *whole* binaries. Thus, it might be hard to apply in situations where patching was performed.

Structure-based static methods Jacobson et al. [16] attempt to fingerprint binary procedures using the sequence of system calls used in the procedure. This approach is unstable to patching, and is only applicable to procedures which contain no indirect calls or use system calls directly.

Smith and Horwitz [31] recognized the importance of statistical significance for similarity problems, yet their method is geared towards source-code and employs n-grams, which were shown ([12]) to be a weak representation for binary similarity tasks.

The authors of [26] show an interesting approach for finding similarity using expression trees and their similarity to each other, but this approach is vulnerable to code-motion and is not suited to cross-compiler search as the different compilers generate different calculation "shapes" for the same calculation.

Detecting software plagiarism `Moss` [8] (Measure Of Software Similarity) is an automatic system for determining the similarity of programs. To date, the main application of Moss has been in detecting plagiarism in programming classes. `Moss` implements some basic ideas that resemble our approach: (i) it decomposes programs and checks for fragment similarity and (ii) it provides the ability to ignore common code. However, in `Moss` the code is broken down to lines and checked for an exact syntactic match, while `Esh` decomposes at block level and checks for semantic similarity. Furthermore, in `Moss` the ignored common code must be supplied by the user as the homework base template code, which is expected to appear in all submissions, while `Esh` finds common strands by statistical analysis. All the ideas implemented in `Moss` are preliminary ideas that bear resemblance to ours but are not fully developed in a research paper or evaluated by experiments over binaries.

8. Conclusions

We presented a new statistical technique for measuring similarity between procedures. The main idea is to decompose procedures to smaller comparable fragments, define semantic similarity between them, and use statistical reasoning to lift fragment similarity into similarity between procedures. We implemented our technique, and applied it to find various prominent vulnerabilities across compilers and versions, including `Heartbleed`, `Shellshock` and `Venom`.

Our statistical notion of similarity is general, and not limited to binaries. We applied it to binaries where the utility of other techniques (that use the structure of the code) is severely limited. In this paper, we used our notion of similarity for retrieval. In the future, we plan to investigate the use of our technique for clustering and classification.

Acknowledgment

We would like to thank Michal Irani for lively discussions, and for permitting the use of figures from [10]. We would like to thank Armando Solar-Lezama, our shepherd, for helping us improve and clarify the paper. We would like to thank Noam Shalev for his inspirational remarks.

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC-COG-PRIME and the Israel Ministry of Science and Technology, grant no. 3-9779.

References

[1] Clobberingtime: Cves, and affected products. `http://www.kb.cert.org/vuls/id/852879`.

[2] Gnu coreutils. `http://www.gnu.org/software/coreutils`.

[3] Heartbleed vulnerability cve information. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`.

[4] Hex-rays IDAPRO. `http://www.hex-rays.com`.

[5] Smack: A bounded software verifier for c programs. `https://github.com/smackers/smack`.

[6] zynamics bindiff. `http://www.zynamics.com/bindiff.html`.

[7] zynamics bindiff manual - understanding bindiff. `www.zynamics.com/bindiff/manual/index.html#chapUnderstanding`.

[8] Aiken, A. Moss. `https://theory.stanford.edu/~aiken/moss/`.

[9] Barnett, M., Chang, B. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures* (2005), pp. 364–387.

[10] Boiman, O., and Irani, M. Similarity by composition. In *NIPS* (2006), MIT Press, pp. 177–184.

[11] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. Bap: A binary analysis platformIn *Proceedings of the 23rd International Conference on Computer Aided Verification* (2011), CAV'11, Springer-Verlag, pp. 463–469.

[12] David, Y., and Yahav, E. Tracelet-based code search in executablesIn *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), PLDI '14, ACM, pp. 349–360.

[13] Egele, M., Woo, M., Chapman, P., and Brumley, D. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* (2014), pp. 303–317.

[14] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (1987), 319–349.

[15] Hawblitzel, C., Lahiri, S. K., Pawar, K., Hashmi, H., Gokbulut, S., Fernando, L., Detlefs, D., and Wadsworth, S. Will you still compile me tomorrow? static cross-version compiler validation. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013* (2013), pp. 191–201.

[16] Jacobson, E. R., Rosenblum, N., and Miller, B. P. Labeling library functions in stripped binariesIn *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools* (2011), PASTE '11, ACM, pp. 1–8.

[17] Khoo, W. M., Mycroft, A., and Anderson, R. Rendezvous: A search engine for binary codeIn *Proceedings of the 10th Working Conference on Mining Software Repositories* (2013), MSR '13, IEEE Press, pp. 329–338.

[18] Kleinbaum, D. G., and Klein, M. *Analysis of Matched Data Using Logistic Regression*. Springer, 2010.

[19] Lahiri, S. K., Sinha, R., and Hawblitzel, C. Automatic root-causing for program equivalence failures in binaries. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I* (2015), pp. 362–379.

[20] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* (2004), IEEE, pp. 75–86.

[21] Leino, K. R. M. This is boogie 2. `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`.

[22] Ng, B. H., and Prakash, A. Expose: Discovering potential binary code re-use. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual* (July 2013), pp. 492–501.

[23] Partush, N., and Yahav, E. *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, ch. Abstract Semantic Differencing for Numerical Programs, pp. 238–258.

[24] Partush, N., and Yahav, E. Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014* (2014), pp. 811–828.

[25] Pewny, J., Garmany, B., Gawlik, R., Rossow, C., and Holz, T. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 709–724.

[26] Pewny, J., Schuster, F., Bernhard, L., Holz, T., and Rossow, C. Leveraging semantic signatures for bug search in binary programsIn *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACSAC '14, ACM, pp. 406–415.

[27] Ramos, D. A., and Engler, D. R. Practical, low-effort equivalence verification of real codeIn *Proceedings of the 23rd International Conference on Computer Aided Verification* (2011), CAV'11, Springer-Verlag, pp. 669–685.

[28] Rosenblum, N., Miller, B. P., and Zhu, X. Recovering the toolchain provenance of binary codeIn *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), ISSTA '11, ACM, pp. 100–110.

[29] Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D. J., and Su, Z. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009* (2009), pp. 117–128.

[30] Sharma, R., Schkufza, E., Churchill, B., and Aiken, A. Data-driven equivalence checkingIn *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications* (2013), OOPSLA '13, ACM, pp. 391–406.

[31] Smith, R., and Horwitz, S. Detecting and measuring similarity in code clones. In *Proceedings of the International Workshop on Software Clones (IWSC)* (2009).

[32] Swamidass, S. J., Azencott, C., Daily, K., and Baldi, P. A CROC stronger than ROC: measuring, visualizing and optimizing early retrieval. *Bioinformatics 26*, 10 (2010), 1348–1356.

[33] Weiser, M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.* (1981), pp. 439–449.