Similarity of Binaries through re-Optimization

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 present a scalable approach for establishing similarity between stripped binaries (with no debug information). The main challenge in binary similarity, is to establish similarity even when the code has been compiled using different compilers, with different optimization levels, or targeting different architectures. Overcoming this challenge, while avoiding false positives, is invaluable to the process of reverse engineering and the process of locating vulnerable code.

We present a technique that is scalable and precise, as it alleviates the need for heavyweight semantic comparison by performing *out-of-context re-optimization* of procedure fragments. It works by decomposing binary procedures to comparable fragments and transforming them to a *canonical, normalized form* using the compiler optimizer, which enables finding equivalent fragments through simple syntactic comparison. We use a statistical framework built by analyzing samples collected "in the wild" to generate a global context that quantifies the significance of each pair of fragments, and uses it to lift pairwise fragment equivalence to whole procedure similarity.

We have implemented our technique in a tool called `GitZ` and performed an extensive evaluation. We show that `GitZ` is able to perform millions of comparisons efficiently, and find similarity with high accuracy.

CCS Concepts • **Theory of computation** → **Program analysis**; • **Hardware** → *Emerging languages and compilers*; • **Software and its engineering** → *Source code generation*

Keywords static binary analysis; statistical similarity; binary code search

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062387

1. Introduction

A known challenge facing a security researcher when reverse engineering a stripped binary library or executable is identifying procedures from known libraries within the binary. Countless hours are wasted analyzing code that originates from pre-analyzed or standard libraries. This is due to the fact that the source code gets ported, modified and compiled using various combinations of compilers and optimization flags, and targeting different (CPU) architectures. Any slight change in the compilation process leads to vast differences in the assembly code, rendering the researcher powerless to use any pre-existing knowledge.

The situation is exacerbated when one is trying to locate code vulnerable to a newly discovered 0-day attack. This vulnerable code may have been embedded in countless binaries, being executed on various devices in the organization, and time is of the essence. One notable example is the recently found `Shellshock` [5] vulnerability, which was undiscovered for 20 years and got ported into various versions of `Unix` operating systems, including Apple's `OSX` and `Ubuntu`. This vulnerability even affects jail-broken ARM based iOS devices. The discovery required security researchers in organizations to meticulously search for all devices that may be running the vulnerable procedure, and examine those binaries – an insurmountable task when binaries have been stripped, which is the common case.

Problem definition Given a query procedure q and a large corpus T of (target) procedures, we aim to produce a quantitative measure which defines the similarity of q and each of the procedures $t \in T$ in an overall ranking. Our goal is to give high ranking to procedures in T that originate from the same source code as q, but have been compiled with (any combination of) a different compiler, different optimization flags or a different target architecture. To adhere to real-world scenarios, the result must be produced over stripped binaries with no debug information, and produce a low false-positive rate while maintaining scalability.

Existing techniques To the best of our knowledge, all previous work in binary code search [11, 7, 23, 13, 27, 10, 15] suffers from one or more of the following drawbacks: (i) it does not address the stripped, cross-compiler, cross- optimization and cross-architecture scenario; (ii) it suffers from

a high false positive rate (iii) is does not scale to millions of comparisons; or (iv) it requires dynamic analysis in addition to static analysis. In Sec. 5 we further elaborate on these approaches, as well as address work in related fields, including equivalence checking [29, 20, 26], semantic differencing [25, 24], translation validation [31] and compiler bug-finding [16].

1.1 Our Approach

We propose a new approach to finding similar procedures in stripped binaries. Our search process is accurate and scalable, and works across different compiler vendors, optimization levels and architectures. Our approach is based on the following key steps:

Fragmenting procedures to comparable units We decompose binary procedures into "strands" – data-flow slices of basic blocks [10], and use them as the basic comparable unit for whole procedure similarity.

Finding strand equality through re-optimization We propose a novel "out-of-context re-optimization" technique, which captures procedure semantics through the optimization of its strands. The way procedures are translated to machine code varies widely across different architectures, compilers and optimization levels (Fig. 1(i)). To efficiently and accurately overcome syntactic difference, we capture procedure similarity through the (re-)application of the *compiler optimizer* over its strands, bringing them to a *canonical normalized form* (canonical with respect to the optimizer), thus identifying syntactically different yet semantically equivalent strands. This alleviates the need for heavyweight semantic tools (e.g., a rewrite engine [11] or SMT solver [10]) and allows our approach to scale.

Gaining perspective with a global context To achieve high precision with a very low false positive rate (which is key to our vulnerability search setting), we identify and reduce the significance of common strands – those strands which originate from style, language, compiler or architecture artifacts (e.g. stack handling). We estimate each strand's importance using a global context – a statistical framework based on an approximation of all binary procedures, built by crawling a corpus containing millions of strands from hundreds of thousands of procedures found "in the wild".

Main contributions In this paper we make the following contributions:

- We propose a new representation geared towards cross-{compiler, optimization, architecture} binary code similarity, based on decomposing binary procedures into strands, canonicalizing and normalizing them. This out-of-context re-optimization allows precise similarity matching at scale. Our decomposition is lifter-agnostic, i.e., can be applied to many intermediate representations of assembly (Section 3.1).
- We demonstrate how our representation integrates into building a blazing-fast similarity computation engine, based on hashed strands and employing a statistical framework from a limited sample of binaries, to improve accuracy (Section 3.2).
- We implemented a prototype of our approach in a tool called `GitZ`, which demonstrates our translation and re-optimization process on `x86_64` and `AArch64` binaries. `GitZ` translates binaries from various architectures using the `VEX-IR` representation and features a newly implemented translation engine from `VEX-IR` to `LLVM-IR`, which allows for re-optimization using the `LLVM` optimizer. Our approach enabled `GitZ` to harness powerful many-core setups to compute procedure similarity accurately, at scale (Section 3.2 and Section 4.3).
- We show an extensive evaluation of `GitZ` in its different use cases using a diverse and challenging dataset of hundreds of thousands. We examine cross-{compiler, optimization, architecture} search scenarios independently and together to evaluate the challenge presented by each vector of the problem. We also examine the effect and contribution of the components of our approach. We compare the results to state-of-the-art techniques and show that we produce more accurate results, with an order-of-magnitude speedup (Section 4).

2. Overview

In this section we informally describe our approach using an example found during our evaluation.

2.1 The Challenge in Establishing Binary Similarity

Fig. 1 shows our similarity method at work when applied to two similar computations, taken from the `dtls1_buffer_message` procedure in `d1_both.c`, which is a part of the OpenSSL code package. These computations were created by compiling the said procedure in two different setups, using:

- `gcc` version 4.8, targeting the `AArch64` (64-bit ARM) architecture, and optimizing at `-O0` optimization level. The processing of this computation is presented in the upper half of Fig. 1, and marked (A).
- `icc` version 15.0.3, targeting the `x86_64` Intel architecture, and optimizing at `-O3` optimization level. The processing of this computation is presented in the lower half of Fig. 1 and marked (B).

Fig. 1 is divided into four columns. The first column (left-hand side), marked (i), shows snippets from the binaries created by the compilation setups above, and the other three (ii, iii and iv) show the effects of our method's processing steps on them.

Syntactically different yet semantically equivalent code

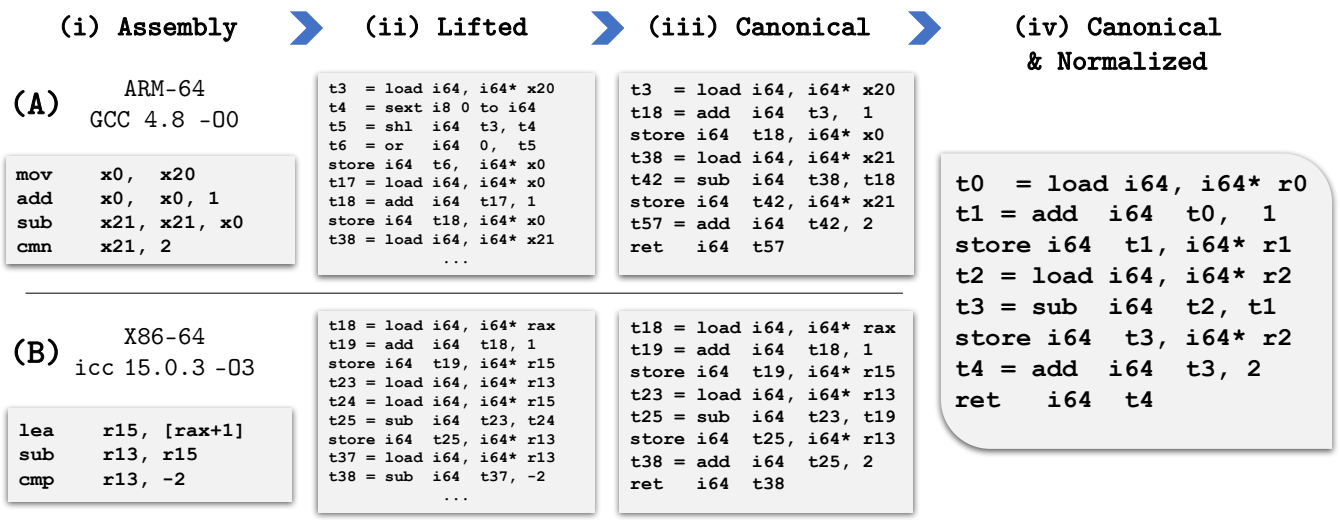

Figure 1. Applying canonicalization, normalization on *strands* taken from our motivating example (`OpenSSL`'s `dtls1_buffer_message` procedure in `d1_both.c`). Only the application of both techniques results in equivalence.

The assembly snippets in Fig. 1(i) perform *exactly* the same computation, yet understanding this requires some familiarity with both architectures. The computations use two values that were generated beforehand and stored in two registers, with an additional, temporary register: the constant 1 is added to the value stored in one of the registers, and the result is stored in the temporary register. The sum is then subtracted from the third register's value. Finally, the computation sets a boolean flag value according to the result of comparing the subtraction's result with -2.

The two computations are performed in different ways, making them hard to compare syntactically. This distinction stems from the different compilation processes, which introduce variance due to many factors, including but not limited to:

Arbitrary register use: The inputs for the computation are stored in three different sets of registers: `x0`, `x20` and `x21` in (A) and `rax`, `r15` and `r13` in (B). The register selection process of the compiler is driven by various heuristics and intricate code pass specifics, which results in a fairly arbitrary selection. Under certain scenarios, even well-known conventions like using `rbp` for the stack frame head are not adhered to by the compiler.

Cross-optimization variance: Another source for syntactic difference in Fig. 1 is the variance in the way code is produced for different optimization purposes. The `-O0` code generated by `gcc` in (A) contains a move and an addition operation, which could have been easily been united to one instruction–`ADD x1,x20,#1`. On the other hand, the `CLang` generated code (B) demonstrates the use of the `lea` instruction to perform a binary arithmetic operation and put its result in a third register without causing other side-effects (as Intel instruction set architecture (ISA) does not support three-address-code instructions like ARM).

Different instruction selection: The snippets from Fig. 1 check whether the input values are equal using different instructions: `cmn` in (A) and `cmp` in (B). `cmn` has similar semantics to the better known `cmp` instruction, yet the former uses addition instead of subtraction to check for equality. This change causes the comparison to be performed against the constant 2 instead of -2.

These variations, found in a short simple computation, demonstrate the challenge in establishing similarity in binary code. Many more variations can be introduced by the application of different optimization levels, using different compilers, or by targeting different machine architecture. In this work, we propose a new approach for overcoming these variations, while finding a new sweet spot between scalability and accuracy.

2.2 Representing Procedures in a Decomposed, Canonicalized & Normalized Form

Lifting binaries to intermediate representation We adapt and use existing techniques for lifting binary procedures into `LLVM-IR`. This standard representation is beneficial in that: (1) basing our similarity method on `LLVM-IR` allows it to be architecture-agnostic, and (2) the `LLVM-IR` format and accompanying tool-suite is well documented, well maintained and has a plethora of tools for creation, translation and manipulation. Since there are no tools which offer support for translating binary code from *multiple architectures* to `LLVM-IR`, we implemented a translator for the `VEX-IR` [22] to `LLVM-IR`. `VEX-IR` is a well-used and robust intermediate representation (IR) for representing binary code from many architectures.

Fig. 1(ii) shows the result of lifting the assembly snippets in Fig. 1(i) into `LLVM-IR`.

Decomposing procedures to strands We build upon previous work [10], and use a similar decomposition, where

the procedure's basic blocks are further decomposed into *strands* – data-flow dependency chains within the scope of a basic block.

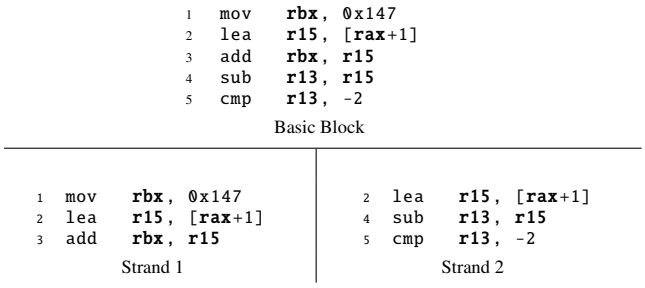

Figure 2. Basic block (top) and extracted strands (bottom) for motivating example (`OpenSSL`'s `dtls1_buffer_message` procedure) compiled to `x86_64` Intel architecture assembly, by `icc` version 15.0.3, optimizing at `-O3` optimization level.

Fig. 2 depicts (top part) a basic block taken from an `icc` compilation of our motivating example, and (bottom part) the two strands extracted from it. A strand is the series of operations leading to the computation of a value within the scope of the basic block. Strand 1 holds the instructions needed to compute the values of `rbx` (and only them) and Strand 2 holds the instructions for computing the comparison result in `cmp r13, -2`. Note that instruction 2 participated in both computations and thus appears in both strands. The pseudo-code for the extraction process is further shown in Algorithm 1. Strand 2 is used as the starting point of the canonicalization and normalization depicted in Fig. 1(B).

Examining the lifter outputs in this example, even for short snippets, exposes additional challenges introduced by the lifting process, when attempting to establish similarity: (1) In (A), a very complex set of instructions are produced to express the register move operation (`mov x0,x20`), which include the `or` and `shift` operations. (2) In both examples redundant `load` operations are performed, loading `t39` in (A) and `t24` in (B) instead of using `t17` and `t19` respectively.

Also note that the modeling of `cmn` uses an `add` instruction, while the `cmp` is modeled using `sub` (as they should be).

Canonicalizing strands Fig. 1(iii) shows the benefits of canonicalizing the strands: (1) All of the lifter-imposed changes (the complex move operation modeling and redundant loads) are reverted, (2) the expression is canonicalized to perform the addition with the constant first and the result is put in the register before the subtraction, and (3) the comparison is canonicalized to the simple addition with a positive constant (instead of subtracting with a negative).

Note that this canonicalization step also acts as a *re-optimization* for code which might not have been optimized before. In our case the `mov,add -> add` optimization was performed. It is also worth nothing that the Intel assembly code was changed to use an instruction which the original architecture *does not offer* (`cmn`).

Canonicalization is an important step in finding the equivalence of the snippets from Fig. 1, yet further differences in register selection cause the optimized strands not to match.

Normalized form Fig. 1(iv) shows the last step of our process, and the benefits of the normalization step: the specific name of the register become immaterial, and the temporary values are renumbered to offset the fact the they were initially part of a bigger basic block and that some of the temporary values were removed as they were a part of a redundant operation.

In this final form the two strands are *syntactically equivalent*. Transforming strands to a representation where *semantic equivalence is captured by syntactic equality* instead of using heavyweight theorem provers or dynamic analysis is crucial to allowing our approach to *scale*.

Scalable search using hashed canonical strands Canonicalization and normalization of procedure strands are performed on each procedure independently, followed by hashing of the textual representation of the strands. The set of strand hashes (i.e., a set of numbers), denoted $R(p)$, is then used to represent the procedure, p, in the comparison stage. Comparing strand hashes allows our comparison engine to achieve superior performance, and minimize memory usage.

Determining the statistical significance of a strand Another important component of our approach is the use of a statistical framework that determines the relevance of each strand in establishing similarity. The goal is to distinguish relevant strands attributed to procedure semantics as it appears in the source code, from other strands which are artifacts of targeting a specific architecture or using a specific compiler pass. The statistical reasoning has a twofold effect, as it also offsets some overmatching which may occur due to canonicalization and normalization.

The similarity score of two procedures, q and t, is based on strands which appear in both q and t, i.e., $R(q) \cup R(t)$. The statistical framework factors in the common appearance rate of each strand, denoted by $Pr(s)$, into the similarity score, in an inverse manner. The significance of a strand and its subsequent contribution to the similarity score is the inverse of its probability, s.t. rare strands ($Pr(s)$=0) contribute greatly to similarity where common strands ($Pr(s)$=1) contribute very little.

We determine the significance of a strand s using its common appearance rate, in the (theoretical) set of *all binary procedures* (in existence) $\widetilde{W}$ as follows:

$$Pr_{\widetilde{W}}(s) = \frac{|\{p \in \widetilde{W} | s \in R(p)\}|}{|\widetilde{W}|} \quad (1)$$

Estimating a global context through crawling binaries "in the wild" As computing $\widetilde{W}$ is intractable, our statistical framework is based on sampling a limited number of binaries "in the wild", thus creating an estimation of the global context, which we demote by P. We use P to approximate

the global context $\widetilde{W}$. P can be gathered *offline* (regardless of the query and target procedures being compared), and we show that a relatively small number of procedures (1K) is sufficient for achieving high accuracy in our approach. Given P, we compute the similarity of a query q and a target t, denoted $S(q,t)$ as follows:

$$S_P(q,t) = \sum_{s \in (R(q) \cup R(t))} \frac{|P|}{f(s)} \quad (2)$$

The similarity score between q and t is the sum of the inverse frequency ($f(s)$) of all the strands *shared* between q and t, normalized by the number of unique strands in P. The composition of Eq. 2 is further explained in Sec. 3.2 by Eq. 3 and 4.

3. Algorithm

3.1 Representing Binary Procedures with Canonicalized Normalized Strands

Lifting opaque binary procedures to IR The first step used by most tools dealing with binary executables is to lift the procedures into some IR. This move allows the tool builder to focus on the semantics expressed in the binary instructions and not on the way they were emitted by the compiler or arranged by the linker. Prominent frameworks for performing these steps are `Mcsema` [4], Binary Analysis Platform (`BAP`) [9], and `Valgrind` [22]. The first uses `LLVM-IR`, while the other two each created an IR (`BIL` for `BAP` and `VEX-IR` for `Valgrind`) for their own purpose. It should be stated that none of these frameworks (nor any other frameworks we encountered) attempt to perform decompilation of the binary code, but rather fully represent the binary instructions' semantics. This is done by representing the machine state using variables, and translating the machine instructions to operations on these variables, according to the machine specification.

This lifting process works by translating each assembly instruction in the procedure into the IR, which explicitly specifies how it affects the machine's memory and registers, including the flags.

Lifted IR inadvertently inhibits similarity

Implementing the lifting process requires an immense amount of work, as there are many machine instructions, some of which cause intricate side-effects and are dependent on other elements of the machine state. To assuage some of these difficulties, the lifting process does not concern itself with being minimal or optimal in any way, and instead focuses on capturing the precise semantics.

Fig. 3 shows three simple examples demonstrating this behavior. In Fig. 3(a) we see a simple move instruction in 64-bit ARM, and Fig. 3(b) shows the lifted `VEX-IR` for it. The figure illustrates how a fairly trivial instruction is modeled using several (relatively complex) arithmetic operations, and using three redundant temporary values (`t14`,`t15` and `t16`). In Fig. 3(c) we see a simple add instruction in 64-bit `x86` assembly, and Fig. 3(d) shows the lifted `BIL` code for it. This lift operation does not use the fact that one of the arguments is a constant, and creates a redundant temporary value (`T2`) for it. This value is then used in subsequent addition operation, even though the `BIL` allows for an addition between a temporary and a constant. Fig. 3(e) displays another simple instruction in 64-bit `x86` assembly, this time a 32 bit subtraction, and Fig. 3(f) shows the lifted `LLVM-IR` code created by `Mcsema`. The created IR again tries to model a more complex computation, this time an unsigned subtraction with overflow, which returns a `struct` containing the subtraction's result and the overflow indication. This is performed even though the next assembly commands do not check the overflow flag; accordingly, the lifted `LLVM-IR` will not extract the second part of this `struct`.

For some use cases, the variance and redundancy of the produced IR are immaterial. However, in the context of program similarity, as shown in Fig. 1(ii), they are damaging, and become disastrous when combined with the other challenges of finding similarity between different architectures and optimization levels.

From lifted binaries to `LLVM-IR` strands The aforementioned lifting tools were created with different design goals in mind: `Mcsema` focused on lifting in a way that allows transformations on the intermediate code such as obfuscation and adding security mechanisms. `BAP` is geared towards performing binary analysis, and `VEX-IR` was built for instrumentation in `Valgrind` dynamic analyses. Furthermore, each tool became specialized at handling specific combinations of the target Operating System (OS) and architectures specifically: `Mcsema` works well on Windows binaries, `BAP` contains a very accurate representation of the `x86` flag computation and `VEX` is adept at generalizing multiple architectures.

We used `VEX-IR` in our prototype, by virtue of its ability to lift binaries from both Intel and ARM architectures and support float instructions. The actual lifting was performed using the `pyvex` library which is part of the `angr.io` binary analysis platform [32]. Regardless of this prototype implementation choice, we wanted our method to be lifter-agnostic, and so we based our representation on `LLVM-IR`, and implemented a translator from `VEX` to `LLVM-IR`. We selected `LLVM-IR`, in view of its stability, extensive set of auxiliary tools and its support for multiple platforms.

Following the lifting and translation process to `LLVM-IR`, our main goal is establishing a procedure representation what will help us find similar procedures.

We start this process by adapting the strand creation process described and used in [10]. This process begins by creating a control flow graph (CFG), and then *slicing* [35] the basic blocks (the nodes of the CFG). A strand is the list of all of the instructions from the basic block that affect the computation of a specific value. The creation process results in a

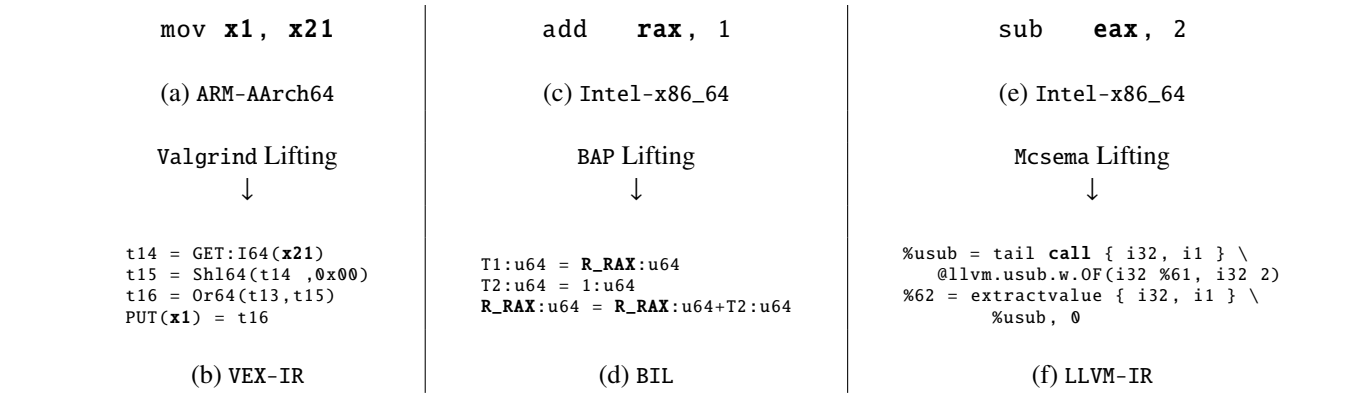

Figure 3. Three examples of binary to IR lifting outputs containing redundant operations

list of strands covering the basic block's instructions, and in turn the entire procedure.

Note that unlike [10], our method does not require the use of (costly) SMT solvers allowing us to use an LLVM-IR representation without the need to translate to BoogieIVL [21].

Algorithm 1 shows the pseudo-code for the strand extraction process. The Ref,Def functions return the set of variables referenced or defined by an instruction. The algorithm iterates over the block instructions, gathering strands according to def-use chains, until all instructions are covered. Note that each created slice is actually a static slice of basic-block, where the sliced variable is not further used in the block.

Algorithm 1: Basic Block to Strands

Input: *bb* - A basic-block as a list of instructions
Output: *S* - *bb*'s strands

```
1  uncovered ← [0, 1, ..., |bb| − 1]; S ← [];
2  while |uncovered| > 0 do
3      last ← uncovered.pop_last ();
4      strand ← [bb[last]];
5      used ← Ref(bb[last]);
6      for i ← (last − 1)..0 do
7          needed ← Def(bb[i]) ∩ used;
8          if needed ≠ ∅ then
9              strand.append (bb[i]);
10             used ∪= Ref (bb[i]);
11             uncovered.remove (i);
12     S.append (strand);
```

Canonicalizing LLVM-IR strands Our notion of procedure similarity depends on our method's ability to detect similar strands across different procedures. This requires our method to overcome three major difficulties: (i) the compiler-imposed changes to the way the strand's semantics are represented, e.g. optimizations for size or runtime and control flow manipulation, (ii) the machine-imposed constraints, e.g. the number of general-purpose registers available, as well as the expressiveness of the instructions-set and (iii) lifter-related changes similar to those detailed earlier in this section.

The basis for establishing similarity using strands is finding *semantically equivalent* strands, where this equivalence accommodates for register renaming and other compiler or architecture artifacts; e.g., `r12 + (rax * rbx)` will be equivalent to `x2 + (x4 * x7)`.

One way to find semantically equivalent strands while avoiding performance-heavy solvers is to move to a canonical form. This transformation should funnel all semantically-equivalent strands to the same *syntactic* representation.

Fortunately, the problem of canonicalizing expressions is well researched and implemented in modern compilers to allow common-subexpression elimination and other optimizations at different levels (procedure-wide or at the basic block level) as part of compilation-time optimizations.

As we are using LLVM-IR to represent our strands, an obvious choice for a canonicalizer is the CLang optimizer (opt). Internally we represent each strand as an LLVM procedure, which accepts as input all the registers partaking in the computation. To allow us to harness the optimizer to our goal, we perform two simple transformations to the LLVM procedure: (i) change the machine register's representation to global variables and (ii) add an instruction returning the strand's value. Note that these changes are required only because the strand was extracted from its context in the binary procedure.

The important optimizer phases we employ are 'common subexpressions elimination' and 'combine redundant instructions' (activated by `-early-cse` and `-instcombine` respectively). The passes perform canonicalization of the expression under specific pre-defined rules. These include re-associating binary operations, grouping subsequent additions, converting multiplications with power-of-two constants to shifts, and many more.

One example of this process is shown in the move from (ii) to (iii) in Fig. 1.

For more information on these optimization phases, we refer the reader to the LLVM documentation [3] with a friendly

warning that a full understanding of this process is possible only by closely examining `opt`'s source code.

Out-of-context re-optimization is crucial to establishing similarity Refining the representation of our strands by moving to a canonical form is an important step in our similarity search method. Its important to note that optimizing entire procedures or basic blocks has little effect in most scenarios because multiple computation paths might be intertwined. This is a major challenge in establishing binary similarity (as demonstrated in Fig. 1). *Re-optimizing* the code allows us to find similarity between unoptimized (e.g., `-O0`) assembly code generated for one architecture, with heavily optimized code for a *completely* different machine architecture. Furthermore, the fact that we are targeting the code to the *same* 'virtual' machine (the `LLVM` abstraction) helps close the gap between the different architectures for comparison purposes.

Moving to a normalized strand representation During the canonicalization step performed by the compiler, the strand (in our case a procedure with a single basic block) is represented by a directed acyclic graph (DAG) which stores the expression. Even though comparing DAGs is possible, we wanted to simplify our representation to some kind of textual form, allowing for fast and simple comparison. This is accomplished by using `opt` to output a linearized version of the computation's DAG. To finalize the transformation which eliminates the origin and compilation choices made in the creation of the binary code, the final refinement to our representation is *normalizing* the strands. This is done by renaming all symbols in the strand, i.e., its registers and temporary values, into sequentially named symbols. This step is crucial for cross-architecture comparison, as the names of the specific registers used in a given computation have *nothing to do* with its actual semantics, and are completely different between architectures.

3.2 Scalable Binary Similarity Search

Using hashed strands to detect similarity The steps performed to canonicalize and normalize the strands allow for efficient comparison, as they replace complex and heavy *semantic matching* with *syntactic equality* to establish procedure similarity. Another important advantage for this transition is that we index the procedure as the set of `MD5` *hashes* and efficiently compare them in this representation. This allows for fast comparison and a reduced memory footprint. Thus, given a procedure p, we denote its representation, $R(p)$, as the set of `MD5` hash values over the canonicalized and normalized strands of the procedure:

$R(p) = \{\texttt{MD5(Canonicalize\&Normalize}(s_p)))|s_p \in p\}$.

A basic notion of the similarity between a given query and target, q and t, can be achieved based on the intersection of their (hashed) representation denoted $M(q,t) = R(q) \cap R(t)$, yet this similarity metric doesn't perform well in practice, as shown in Sec. 4.5. One reason is that some of the matched strands are *not relevant*, as they are compilation artifacts and have *nothing to do* with the semantics expressed by q (or t).

Accounting for the binary creation process As our evaluation will show, using a certain compiler or targeting a specific architecture in the procedure compilation and assembly generation process adds non-semantic related artifacts in the generated assembly, beyond the original content of the source code. These artifacts might be a side effect of accessing the machine's memory through a certain construct (e.g., a stack), or be related to a specific optimization pattern implemented by the compiler. Previous work, [30] for example, shows that these artifacts can even be used to detect the toolchain used to create the binary.

Knowing the exact origins of every query and target procedure in the corpus might have allowed us to cancel out these artifacts, but retrieving this information is complex and imprecise when working on stripped executables. We thus need to find some other way to separate the semantic strands, which originate from the source code of the query, from the unrelated artifact strands, which can be attributed to the binary creation process.

One relatively simple way to separate the strands is to apply the assumption that a common strand, which appears in many procedures, carries less importance than a rare strand. In general we can define $Pr(s)$ as the probability that a strand s will appear "at random". A more appropriate approach is to limit our probability space by two factors: (i) strands that are lifted from real binaries, and specifically from the group of binaries which are targeting one of the architectures our process can lift, denoted W and $\widetilde{W}$ respectively, and (ii) strands that are canonicalized and normalized.

As defined in (1), we determine $Pr_{\widetilde{W}}(s)$ for a given strand s and we divide the number of different procedures in which it appears, by the total number of unique strands appearing in all of the procedures in $\widetilde{W}$.

We use this probability to define our similarity metric:

$$S_{\widetilde{W}}(q,t) = \sum_{s \in M(q,t)} \frac{1}{Pr_{\widetilde{W}}(s)} = \sum_{s \in M(q,t)} \frac{|\widetilde{W}|}{|\{p \in \widetilde{W} | s \in R(p)\}|} \quad (3)$$

Using random sampling to approximate strand frequency Calculating $Pr_{\widetilde{W}}$ is not feasible, due to its dependence on possessing $\widetilde{W}$ (although $\widetilde{W}$ is finite at any given time, it constantly grows). Instead we sample a large subset, denoted P, and use it to approximate $Pr_{\widetilde{W}}$:

$$Pr_{\widetilde{W}}(s) \simeq_P \frac{f(s)}{|P|} \quad (4)$$

where:

$$f(s) = \begin{cases} |\{p \in P | s \in R(p)\}| & s \in P \\ 1 & else \end{cases}$$

P is randomly collected from $\widetilde{W}$ using a crawling process that gathers binary procedures "in the wild", and as such contain equal representation of all supported compilers, optimization levels and architectures. Moreover, smaller sizes for P will decrease the memory footprint of our method. We further discuss the composition and size of P in Sec. 4.6. Note that all the calculations regarding the frequency of every strand $s \in P$ and the (constant) $|P|$ can be done *offline*, and using this data will allow our similarity score calculation to be performed in a *global context* yet still scale. Applying Eq. 4 to Eq. 3 will result in Eq. 2, shown before.

We note that $Pr_{\widetilde{W}}$ is an approximation of $Pr_{\tilde{W}}$, which is a probability. Since it is not feasible to compute $\tilde{W}$, we approximate $f(s) = 1$ for strands not collected in P. This prevents $f(s)/|P|$ from being a probability; however it remains a reasonable approximation as these are mostly rare strands that seldom appear in $\tilde{W}$.

Some important properties of our similarity score are:(i) It is symmetric, allowing to cache similarity scores for any pair for future matchings. (ii) The results of similarity searches with the same query, which outputs two ranked lists, can be joined.

Embarrassingly parallelizable approach Our technique is based on pairwise strand hash comparison operations and the application of the (offline) global context to produce a similarity score. Thus we were able to deploy our tool on a powerful many-core setup, where each query-target pair comparison is assigned to a core. The global context is precomputed and loaded by each of the cores interdependently. A pair's similarity score is thus produced by a single core in under a second, on average, independently of other comparisons. The procedure indexing process is parallelizable as well, where each procedure is indexed by a different core. We performed all of our experiments on a machine with four `Intel Xeon` E5-2640 (2.90GHz) processors (72 cores), 368 GiB of RAM, running `Ubuntu 14.04.2 LTS`. Each process uses at most 300MiB during its run. In the machine's view, when all 72 cores are used, the engine uses 23 GiB.

4. Evaluation

In this section we evaluate `GitZ`. Our evaluation aims to answer the following questions:

- How useful is `GitZ` in the vulnerability search scenario (Sec. 4.3)?
- How well does `GitZ` scale (Sec. 4.3)?
- How accurate is `GitZ` and how does it measure (independently and altogether) in each of the problem vectors: different architectures, different compilers and different optimization levels (Sec. 4.1)?
- How does each component of our solution affect the accuracy of `GitZ` (Sec. 4.5)?
- How does `GitZ` measure in comparison to previous work (Sec. 4.3)?

But first we will explain our corpus creation process and design goals, and present our evaluation metric.

4.1 Creating a Corpus to Evaluate the Different Problem Vectors

To perform a through evaluation, we will need to compare binaries where the ground truth is known. To this end we create binaries according to our three problem vectors:

Different architectures As illustrated by our motivating example (Fig. 1), the same source code compiled to different architectures is inherently different. The instruction set is different, and even after lifting to an intermediate representation, the code remains different, due to different paradigms implemented by the architecture. For instance, Intel's `x86_64` architecture allows instructions to operate over the higher 8 bit part of certain registers (`[abcd]h`), while ARM's `AArch64` does not allow accessing data at that resolution, and requires further computation to extract the same value (see another example in Fig. 5). To measure the accuracy of our technique in the cross-arch setting, our corpus included binaries from two widely used architectures: Intel `x86_64` and ARM `AArch64`.

Different compilers Different compilers produce binaries which differ immensely in syntax [12]. Different compilers may use different registers, employ different instruction selection, order instructions differently, structure the code differently, etc. To evaluate our ability to overcome these differences, our test corpus was compiled using prominent compilers from 3 different vendors, with several versions for each compiler.

Different optimization levels Modern compilers apply various optimization methods. For instance, with the `-O1` and `-O2` optimization levels for the `gcc` compiler, as many as 40 different optimization passes are performed [2]. To see whether `GitZ` is able to identify similarity across optimizations, each binary was compiled using each of the optimization flags.

Compilation setups In our evaluation we will use:

- C_{x64} – The set of compilers targeting the Intel `x86_64` architecture containing `CLang` 3.{4,5}, `gcc` 4.{6,8,9} and `icc` {14,15}.
- C_{ARM} – The set of compilers targeting the ARM `AArch64` architecture containing `aarch64-gcc` 4.8 and `aarch64-CLang` 4.0.
- O – A set of optimization levels, `-O{0,1,2,3,s}`.

To this end we have created a utility named `Compilator`, which receives a code package as input and compiles it with each of the configurations from $\{C_{x64} \cup C_{ARM}\} \times O$, resulting in 44 binary versions for each procedure. We used `Compilator` to create ~500K binary procedures from prominent open-source software packages, including OpenSSL, git, Coreutils, VideoLAN, bash, Wireshark, QEMU, wget and ffmpeg. Some packages were intentionally chosen as

			(a)			(b)							
#	CVE	Tool \ Alias	GitZ-500K: Cross-{Comp, Arch, Opt}			GitZ-1500: Cross-Comp			Esh-1500: Cross-Comp			# VEX Strands	#Canon Norm Strands
			#FPs	CROC	Time	#FPs	CROC	Time	#FPs	CROC	Time		
1	2014-0160	Heartbleed	52	.999	15m	0	1	1s	0	1	19h	89	45
2	2014-6271	Shellshock	0	1	17m	0	1	3s	3	.996	15h	233	71
3	2015-3456	Venom	0	1	16m	0	1	1s	0	1	16h	47	20
4	2014-9295	Clobberin' Time	0	1	16m	0	1	2s	19	.956	16h	153	58
5	2014-7169	Shellshock #2	0	1	12m	0	1	2s	0	1	11h	359	104
6	2011-0444	WS-snmp	0	1	14m	0	1	1s	1	.997	10h	65	43
7	2014-4877	wget	0	1	10m	0	1	2s	0	1	15h	214	47
8	2015-6826	ffmpeg	0	1	17m	0	1	1s	0	1	20h	74	30
9	2014-8710	WS-statx	0	1	18m	0	1	2s	-	-	-	104	55

Table 1. CROC and #FP for the vulnerability search experiments: (a) Over 500K procedures, cross-{arch, compiler, optimization} and (b) A comparison of GitZ to [10].

they contained a vulnerability (in a specific version), which was used in the experiments in Tab. 1.

We crawled the corpus to randomly select 1K procedures, and used their strands to build our global context P. Further details regarding the global context composition and size can be found in Section 4.6.

4.2 Evaluation Metric

In our approach, the similarity of a pair of procedures is quantified as a real number value, within the global context. This means that for every query procedure, we produce a ranking of similar target procedures, and thus require a way to evaluate this ranking. We use two metrics: The CROC metric, which is widely used in assessment of early retrieval methods [34] and measures whether there are many false positives at the top of the ranking. We also include the percentage of false positives encountered until all true positives are covered, which is another reflection of the CROC measure. However it is important to note that CROC measures the *rate* of false positives encountered as the threshold increases, and not just their number or percentage. The CROC measure is an adaptation of the Receiver Operating Characteristic (ROC) method for scenarios with huge corpora. In general, the ROC method operates by treating the ranked list as a classifier (i.e., setting a strict threshold and treating all results above it as positive and all below as negative) and measuring its accuracy as follows: $Acc(thresh) = (TP+TN)/(P+N)$ (where TP stands for true positives, TN for true negatives, etc.). The accuracy is then computed over all thresholds found in the ranking and the result is plotted. The area under the curve (AUC) for this graph represents the ROC measure. CROC operates similarly but assigns a higher penalty for false positives. This makes it more appropriate for our scenarios, as a human expert will often need to review the results, and thus a low number of false positives is crucial. It is critical to note that CROC reflects the accuracy across all thresholds. In fact, there is no clear way to set a strict global threshold for classifying a score value as a positive match over all experiments, as similarity scores vary according to query and corpus size. We use Yard-Plot [6], which uses (by default) a magnification factor of $\alpha = 7$, which transforms a FPR of 0.1 to 0.5. Further discussion of the benefits and implementation of this widely used method is provided by Swamidass et al. [34].

4.3 GitZ as a Scalable Vulnerability Search Tool

To evaluate GitZ in the vulnerable code search scenario, we used real-world vulnerabilities and searched for them in our corpus of procedures. The experiment shows how GitZ can be used by a security-savvy organization that wishes to discover whether it might be vulnerable to a newly found 0-day exploit.

Tab. 1(a) details our main experiment, where 9 real-world vulnerable procedures from open source projects are used as queries and searched against our full 500K procedure corpus. The corpus contained 44 true positives (i.e., similar procedures originating from the same source code) for each query. For each procedure the number of false positives, the overall accuracy expressed using the CROC measure, and the overall runtime (rounded) are specified. In all experiments but one, GitZ was able to rank all true positives above all unrelated procedures. For experiment #1 (Heartbleed), 52 false positives were ranked above a true positive (0.0001 FP rate).

Tab. 1(b) further shows a comparison of GitZ to the tool Esh from [10]. We recreated the experiment from [10] (using the publicly available dataset [1]), which includes searching for the vulnerabilities in a corpus of 1500 procedures, across compilers (Esh did not apply to the cross-architecture or cross-optimization level scenarios). The results reported in [10] are marked Esh-1500, and the results for running the same experiment with GitZ appear alongside, marked GitZ-1500. Tab. 1(b) shows that GitZ is able to produce *more accurate results with 0 false positives*, for the same scenario. Furthermore, since Esh relies on a program verifier, its average runtime is 15.3 hours. GitZ, on the other hand, provides a *speedup of 4 orders of magnitude*, from tens of hours to an average run time of 1.8 seconds. Finally, when added *a*

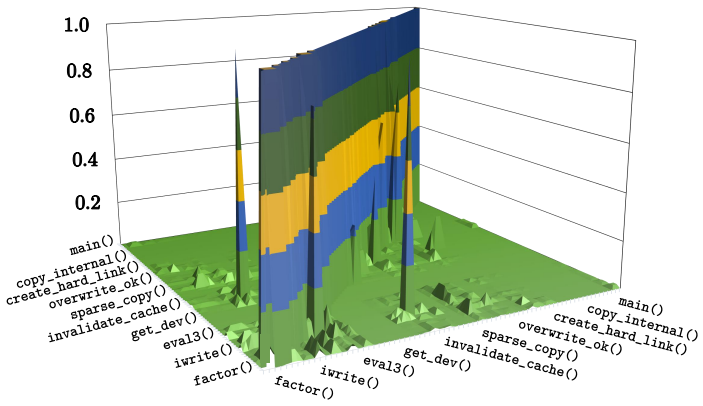

Figure 4. Similarity score (as height) for All v. All experiment. Average CROC: .978, FP rate: .03, Runtime: 1.1h.

Scenario	#	Queries	Targets	CROC	FPr
Cross-*	1	*	*	.977	.03
Cross-Arch,Opt	2	C_{ARM} -O*	C_{x64} -O*	.963	.01
	3	C_{x64} -O*	C_{ARM} -O*		
Cross-Opt, Version	4	gcc$_{x64}$ 4.{6,8,9} -O*	gcc$_{x64}$ 4.{6,8,9} -O*	.999	.001
	5	icc$_{x64}$ {14,15} -O*	icc$_{x64}$ {14,15} -O*	.999	.001
	6	CLang$_{x64}$ 3.{4,5} -O*	CLang$_{x64}$ 3.{4,5} -O*	1	0
	7	gcc$_{ARM}$ 4.8 -O*	gcc$_{ARM}$ 4.8 -O*	1	0
	8	CLang$_{ARM}$ 4.0 -O*	CLang$_{ARM}$ 4.0 -O*	1	0
Cross-Comp x86_64	9	C_{x64} -Os	C_{x64} -Os	.992	.001
	10	C_{x64} -O0	C_{x64} -O0	.992	.001
	11	C_{x64} -O1	C_{x64} -O1	.986	.002
	12	C_{x64} -O2	C_{x64} -O2	.992	.001
	13	C_{x64} -O3	C_{x64} -O3	.992	.001
Cross-Comp AArch64	14	C_{ARM} -Os	C_{ARM} -Os	.988	.002
	15	C_{ARM} -O0	C_{ARM} -O0	.995	.001
	16	C_{ARM} -O1	C_{ARM} -O1	.999	.001
	17	C_{ARM} -O2	C_{ARM} -O2	.995	.001
	18	C_{ARM} -O3	C_{ARM} -O3	.998	.001
Cross-Arch	19	C_{x64} -Os	C_{ARM} -Os	.969	.006
	20	C_{ARM} -Os	C_{x64} -Os		
	21	C_{x64} -O0	C_{ARM} -O0	.977	.004
	22	C_{ARM} -O0	C_{x64} -O0		
	23	C_{x64} -O1	C_{ARM} -O1	.960	.006
	24	C_{ARM} -O1	C_{x64} -O1		
	25	C_{x64} -O2	C_{ARM} -O2	.965	.005
	26	C_{ARM} -O2	C_{x64} -O2		
	27	C_{x64} -O3	C_{ARM} -O3	.975	.004
	28	C_{ARM} -O3	C_{x64} -O3		

Table 2. Accuracy and FP rate for different derivatives of the All v. All experiment.

new vulnerability (CVE 2014-8710) which did not appear in the Esh experiment, GitZ was able to find all true positives with 0 false positives. There is a slight loss in precision for vulnerability #1 (Heartbleed) for the GitZ-500K experiment (compared to GitZ-1500), which is attributed to the more challenging cross-{arch, opt} scenario, and the vastly larger corpus.

The last two columns in Tab. 1 show the number of (unique) raw VEX strands and canonical normalized strands extracted from each procedure. The numbers show the first positive effect of our normalization and canonicalization passes, as they help reduce the procedure representation. These passes further affect accuracy, as explained in Section 2.

4.4 All vs. All Comparison

To better evaluate GitZ's accuracy and performance over these challenging scenarios, we performed an experiment using a subset of 1K procedures, selected at random from our 500K corpus. The experiment was executed in an "All vs. All" setting, where every procedure out of the thousand picked is searched against all others (approximately 1 million comparisons). The experiment's goal was to evaluate our approach, when either the query or the target binaries consist of any and all of the varying architectures, compilers and optimization levels. In this setting, GitZ reports an average CROC accuracy of .978, with an average FP rate of .03. The overall run time for the experiment was 1.1 hours.

Due to the size of the experiment, we cannot include a table showing the result for each of the procedures. Instead, we present results in the form of a surface (Fig. 4) showing the normalized similarity for 100 randomly selected queries.

Fig. 4 shows the normalized similarity result as surface height. Both axes have the same dataset: The list of experiments ordered by name and *grouped together* according to source procedure, i.e., all compilations of the same procedures are coalesced. Important observations are:

(i) The diagonal shows the ground truth (all procedures are matched with themselves) along with other compilations of the same procedure. These can be see as "ridges" along the diagonal "wall". The similarity score for the same procedures compiled differently is, as expected, lower than that of the ground truth and is dependent (for the most part) on how different the architecture, compilers and optimization flags are.

(ii) The surface is symmetrical w.r.t the diagonal. This is expected as our similarity metric is symmetrical i.e. $S(q,t) = S(t,q)$, due to the use of an offline global context.

(iii) Some "spikes" seem out of place. For instance the invalidate_cache() procedure from dd.c seems to match with an unrelated procedure and create a false positive. Upon closer examination, we learn that the matched procedure is iwrite() also from dd.c, where in fact invalidate_cache() *is a callee*. The matching occurs because in that specific compilation, if iwrite(), the callee invalidate_cache() was *inlined*, and the entire procedure body resides inside the binary, thus explaining the match and asserting it as a true positive.

Evaluating all vectors, together and separately Tab. 2 shows the accuracy of GitZ for several combinations of

compilers, optimization levels and architectures for the aforementioned All v. All experiment. These experiments are aimed at answering questions like: (i) "How well does our technique find similarity between less and more optimized code, produced by the same compiler?" (ii) "Which of the search settings (opt, comp, arch) is most challenging?" (iii) "Does optimization level factor when searching across compilers, architectures?"

Tab. 2 presents the results from 28 different settings of the experiment where: (i) the "Scenario" column groups similar experiments, (ii) the "Queries" and "Targets" columns specify the subset of compiler setups used to generate the queries and targets' and (iii) the CROC and FPr specify accuracy and false positive rate.

Similarity within compiler, architecture boundaries: Lines 4-8 of Tab. 2 show that GitZ is adept at finding similarity whenever the code is compiled with the same compiler, but different optimization level, regardless of compiler vendor, or even target architecture. Lines 9-18 show a slight loss of precision, for the cross-compiler (but same architecture and optimization level) scenario. No particular optimization level showed to be more challenging than the others.

Challenging cross-architecture scenarios: Lines 2-3 and 19-28 in Tab. 2 describe the cross-architecture scenario, where the compiled queries targeting the AArch64 architecture are searched for in a corpus of x86_64 targets, and vice versa. Each experiment is represented using two lines in the table to emphasize the distinction made between architectures, i.e. when searching for an AArch64 query, the target corpus included only x86_64 and vice versa. The cross-architecture scenario presents the greatest challenge for GitZ. This challenge stems from different implementation paradigms for various operations and instructions, which result in strand mismatch, even after canonicalization and normalization.

Fig. 5 depicts basic blocks taken (with some changes made for brevity of presentation) from the add_ref_tag() procedure in pack-objects.c, which is a part of version 2.10.0 of the git project. The two blocks perform the same operation – preparing arguments for the procedure call packlist_find(&to_pack, peeled.hash, NULL). Argument preparation is as follows: (i) The address of the to_pack argument is stored into the x0 register in the AArch64 case (Fig. 5(a), lines 1-2), and the edi register in the x86_64 case (Fig. 5(b), line 3); (ii) NULL is stored into the x2 register in the AArch64 case (Fig. 5(a) line 4), and the edx register in the x86_64 case (Fig. 5(b) line 1); and (iii) the peeled.hash field, which belongs to the locally allocated peeled struct, is computed at offset 0x30 of the stack pointer and assigned into x1 in the AArch64 case (Fig. 5(a), line 3), but assigned directly to the rsi register in the x86_64 case (Fig. 5(b), line 2), due to a different memory outline. Note that we intentionally left in the callee's name for clarity; however, procedure names are in no way used to establish similarity in GitZ, as they do not exist in the stripped binary setting.

```
1  adrp    x0, #to_pack
2  add     x0, x0, #to_pack
3  add     x1, sp, #0x30
4  mov     x2, #0
5  bl      packlist_find
```

(a) gcc 4.8 AArch64 -O1

```
1  xor     edx, edx
2  mov     rsi, rsp
3  mov     edi, offset to_pack
4  call    packlist_find
```

(b) gcc 4.9 x86_64 -O1

Figure 5. add_ref_tag() compiled to different archs

Although most differences ((i) and (ii)) are bridged by GitZ, the different memory layout over architectures hinders our approach's matching ability and results in some loss of precision.

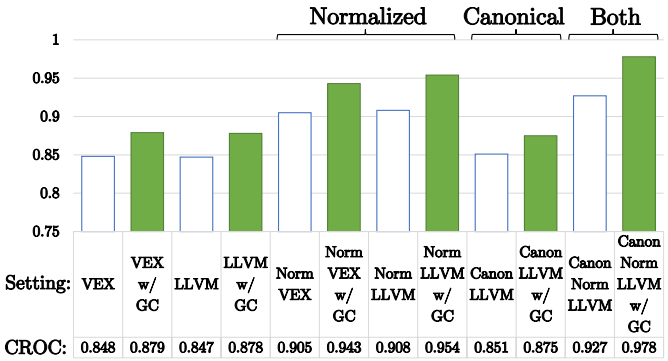

Figure 6. The average accuracy (CROC) of the All v. All experiment, when applying different parts of our solution.

Templating, Vectorization and Inlining: Some language, machine or compiler mechanisms may introduce vast syntactic changes to a procedure, which may also result in different semantics. Generic template procedures, are written as a means to allow functionality over several types, and thus can be compiled to operate over different types. The same template procedure compiled by different compilers will differ even more so when it is compiled to different types. We encountered template procedures in our evaluation; however, as the general algorithm performed by the template procedure is the same, GitZ was able to identify enough shared strands over the differently typed compilations to detect similarity. The same was observed for vectorized vs. non-vectorized assembly code for the same procedure.

Inlining poses a challenge in establishing similarity, as it introduces the code of a different procedure into the body of the query or target. However, in our experiment, it resulted in relatively few wrong matches. Since the core functionality of the caller is preserved in its own strands, these strands were sufficient to capture similarity of true positives, even when the call-graphs vary. In other cases, caller procedures were matched with callees that were inlined, which initially seemed like false positives. To correctly label these results we added a pass flagging caller-callee similarity as a true positive.

4.5 Comparing Components of the Solution

The effect of canonicalization and/or normalization Fig. 6 presents the average accuracy for the All v. All experiment, in terms of the CROC metric (y-axis and below each bar), for different settings (the x-axis) of `GitZ`. The graph shows accuracy results when (incrementally) applying the different components in our techniques, where the leftmost bar represents the accuracy for computing the similarity by simply counting the number of shared strands over the procedures' `VEX-IR` strands, without any canonization or normalization, and without a global context. The rightmost bar shows accuracy when applying our complete approach, which is also the result reported in all previous experiments. Fig. 6 is separated into blank bars and full bars, representing results without and with the application of a global context P.

The following observations can be made from Fig. 6: (i) The use of a global context uniformly increases precision for all settings. (ii) Normalization is vital in achieving syntactic equality, which is to be expected due to the high variance in register (and temporary) names, originating from the architecture. (iii) The canonicalized, normalized scenario (2 rightmost bars) is highly affected by the global context, with a precision gain of 0.051 in CROC, which translates to a substantial false positive rate drop from 0.16 to 0.03. This emphasizes the beneficial dependence of this setting (which is the de facto setting for `GitZ`) on the global context, as normalization and canonicalization group together more strands, thus reducing their significance in $Pr_{\widetilde{W}}(s)$.

4.6 Understanding the Global Context Effect

Evaluating different compositions of P To throughly understand how the global context affects our method's success at finding similar procedures, we experimented with running `GitZ` using different variations of P – the approximation of the global context.

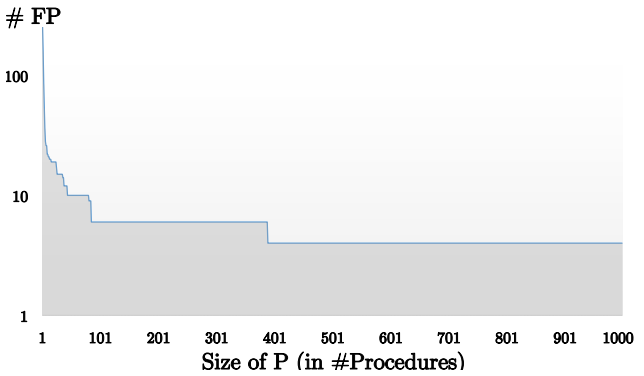

Figure 7. Examining the importance and effect of using the global context by measuring number of false-positives as P grows.

Fig. 7 depicts the average number of false positives, as a function of the size of P, across 5 randomly selected experiments. In each experiment we selected one query from the All vs. All corpus and searched for it within the 1K-sized corpus. Each experiment consisted of multiple runs where a different (gradually growing in size) set of procedures were selected for P and used to perform the similarity score calculation.

Fig. 7 shows how, at the beginning of the experiment, each increase in the size of P is reflected in a major decrease in the number of false positives, and eventually a fix point is reached at around 400 procedures.

Although not expressed in the figure (for clarity and proportion), any further attempts to increase the size of P, even up to the size of our bigger 500K corpus, did not change the number of false positives in our experiments.

Following these results we set our P size to 1K (2.5 times the largest effective size), and composed it by randomly picking a subset of this size from of our 500K corpus. We proceeded to perform *all* of the experiments in this section with the said subset.

Delving into the Composition of the Global Context

After building our global context, P, we more deeply examined its contents and found two dominating and subgroups of strands in it.

Common strands appearing in all architectures: One type of strand which occurs frequently, and in all examined architectures, is the stack-frame setup and destruction in the prologue and epilogue of some (mostly large) procedures. The frame setup is performed by adding and subtracting from `rsp` and `xsp` in the Intel and ARM architectures respectively. Another common strand was the saving and restoring the callee-saved registers. We encountered this type of strand in different variations, some include partial use of the callee-saved registers, some use different stack offsets. Despite these variations, the strands were successfully detected as low significance strands within the global context, due to the optimization and normalization stages. The transformation converted multiple strands of the said type into one syntactic form, across compilers and architectures, marking them as common.

Common strands unique to specific architectures: One reason for `GitZ`'s lower precision in the cross-arch scenario (shown in Tab. 2 and explained above) resides in the inherent differences between the architectures, differences that affect the representation of the global context. For example, one very common instruction encountered in Intel assembly code is `xor some-register,same-register`. This instruction simply puts the value of zero in a register, instead of using `mov some-register, 0`. This is used as a code-size-optimization, as the `xor` operation on any register is represented using two bytes, while moving the immediate zero requires between three and eight bytes (for the instruction itself and the zero immediate). The ARM architecture aligns all instructions to size 4, and therefore such instruction size maneuvers are not performed. Moreover, ARM contains a special 'virtual' register called `zr` (or `r31`), which always

holds the value of zero and alleviates the need for a zero immediate altogether. Furthermore, the ARM architecture supports some very useful instructions *not present* in the Intel architecture. One example is the 'Compare Branch Zero' (`cbz`) instruction, which jumps to a specified offset if the operand register is equal to zero. Several Intel instructions can be used to represent this operation, i.e., `cmp reg, 0; jz offset`. However the flags register will be affected by the `cmp` instructions, essentially creating a different computation. In this scenario the Intel computation will contain a new flag storing the comparison's result, breaking the equivalence, and in turn causing the computation's representation in the strands to diverge and not match.

4.7 Limitations

`GitZ` relies on the premise that the same source code compiled differently will contain the same chains of execution (strands), and these can be extracted and transformed to an equivalent form through normalized canonical strands (Fig. 1) while reducing the importance of common strands using statistical reasoning (Fig. 6). Our evaluation shows that equivalent strands will almost always be found across different compilations. However, similarity indeed cannot be established in some cases, and these become more frequent as the two compilations diverge.

Broken basic blocks: As compilation diverges, the resulting control flow graphs of the procedures differ from one another. Fig. 8 depicts the source code (marked (a)) and the starting basic blocks of three compilations of `wget`'s `ftp_syst()` procedure from file `ftp-basic.c` (marked (b)-(d)). Two of the compilations (marked (b) and (c)) are performed by the same compiler make (`CLang`) with different versions of the compiler (3.4 and 3.5 respectively), while the third (marked (d)) is compiled using `gcc` 4.6. Fig. 8 shows how the more current versions of `CLang` operate similarly, as both versions 3.4 and 3.5 include the call to `free()` as part of the first basic block. The `CLang` compiler (in both versions) recognizes that both branch destinations of the conditional in Fig. 8(a) line 8 arrive at a call to `free()`. `gcc` at version 4.6 is less optimized and does use the fact that `free()` is called in both branches. This difference (among others) results in a lower similarity score (19 mutual strand for `gcc` 4.6 vs. `CLang` 3.5, as opposed to 40 mutual strands for `CLang` 3.4 vs. `CLang` 3.5). This illustrates a limitation of `GitZ`, as *it operates at the boundaries of a basic block*. The decision to break procedures at basic blocks was made due to performance and scalability concerns. Other approaches (e.g. "tracelets" from Yaniv et al.[11]) which employ longer chains of executions can be incorporated into our approach to overcome the limitation, at the price of reduced performance.

5. Related Work

Next, we briefly describe related work that applies to relevant yet different motivations or scenarios.

```
1  uerr_t
2  ftp_syst (int csock, ...){
3    ...
4    /* Send SYST request.  */
5    request = ftp_request ("SYST", NULL);
6    nwritten = fd_write (csock, request,
7                         strlen (request), -1);
8    if (nwritten < 0) {
9        free (request);
10       return WRITEFAILED;
11     }
12   free (request);
13   ...
14 }
```

(a) Source code

```
mov     r14, rsi
mov     r15, rdi
mov     edi, 0D9Eh
xor     esi, esi
call    ftp_request
mov     rbp, rax
mov     ebx, [r15]
mov     rdi, rbp
call    strlen
mov     edi, ebx
mov     rsi, rbp
mov     edx, eax
call    iwrite
mov     ebx, eax
mov     rdi, rbp
call    free
test    ebx, ebx
mov     ebp, 37h
js      loc_B20
```

(b) `CLang 3.5 -O2`

```
mov     r14, rsi
mov     r15, rdi
mov     edi, 0D8Eh
xor     esi, esi
call    ftp_request
mov     rbp, rax
mov     ebx, [r15]
mov     rdi, rbp
call    strlen
mov     edi, ebx
mov     rsi, rbp
mov     edx, eax
call    iwrite
mov     ebx, eax
mov     rdi, rbp
call    free
test    ebx, ebx
mov     ebp, 37h
js      loc_B0D
```

(c) `CLang 3.4 -O2`

```
mov     rbp, rdi
mov     edi, 0F93h
sub     rsp, 28h
mov     r12, rsi
xor     esi, esi
call    ftp_request
mov     rdi, rax
mov     rbx, rax
call    strlen
mov     edi, [rbp+0]
mov     edx, eax
mov     rsi, rbx
call    iwrite
test    eax, eax
mov     rdi, rbx
js      loc_C18
```

```
call    free
...
```

```
call    free
...
```

(d) `gcc 4.6 -O2`

Figure 8. Source code and assembly for different compilations of `ftp_syst()`

Similarity in binaries: Ng and Prakash [23] operate in a similar context of finding code-reuse and IP theft in binaries, using symbolic execution and theorem provers. They lack a cross-compiler evaluation, and use some syntactic heuristics (number of arguments) which affect accuracy. [7] a commonly used commercial tool for comparing whole binary libraries and executables. It performs a many-to-many comparison of all of the procedures inside the binary, and relies on the connections between these procedures (call-graph) as partial evidence of similarity. `BinDiff` ignores procedure semantics altogether [8]. The comparison is heavily based on syntactic heuristics, including procedure names (which are unavailable in stripped binaries), the number of jumps, the number of basic blocks, etc. [11] is also heavily biased towards syntax, and therefore fails to handle differences that result from different compilers, as shown by [10]. Pewny et al. [27] uses sampling of SMT-solver simplified basic blocks, lifted using `VEX`. They report false positives in their results, which can be linked to the sampling based method because it is not countered by a statistical framework to account for spurious matches from commonly found computations. Furthermore, they require a complex BHB process to be performed online on each pair of compared CFGs (after some pruning). Recently the authors of [15] extracted syntactic and structural features from the code, which hinders

their applicability to the stripped binary search scenario. The authors also report a high false positive rate (even for the bug-search scenario).

Dynamic approaches: Egele et al. [13] propose a dynamic method where the procedures are executed using random values and the resulting environment is compared. The reported accuracy is problematic in our scenarios (a tool for a human expert). Aiken et al. [31] find the equivalence of source and machine level code by searching for a simulation relation based on traces of execution. The data driven approach could be applied to our setting of a binary similarity but may not scale due to the computationally heavy search process.

Moreover, dynamic approaches are limited because the executable may not be easily run externally (i.e., it may target a different architecture than the researcher's machine) and may thus require non-trivial execution access to devices (e.g., IoT devices). Also, running (any part) of the possibly vulnerable library may expose the user to risk.

Semantic differencing: Partush et al. [25, 24] produce an abstract representation of program difference for `C` programs with loops. The work is limited by the use of costly abstract domains and does not operate on binaries. SymDiff by Lahiri et al. [20] leverages a program verifier to prove equivalence of whole procedures by translating them to `BoogieIVL`, or produce a counterexample. It has limited handling for loops and will require a translation of binary to full procedures. Furthermore, the use of provers prevents the approach from scaling.

Equivalence checking: Engler et al. [29] present a symbolic execution approach for proving procedure equivalence for finding bugs in different implementations of the same procedures. They work on the source code level and their method is limited to non-looping code. Lahiri et al. [20] apply a recursion rule to verify the equivalence of recursive functions. The rule operates at the level of source code and relies on a theorem prover, which limits its scalability.

Locating compiler bugs: Hawblitzel et al. [16] compare intermediate language (IL) code produced from different compilers to root-cause compiler bugs, and thus focus on full equivalence instead of similarity. This approach is not likely to scale in our setting, as it uses semantic tools for proving equivalence.

Structure-based static methods: The authors of [28] present 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. Jacobson et al. [17] attempt to fingerprint binary procedures using the sequence of system calls used in the procedure. This approach is unstable under patching, and is only applicable to procedures which contain no indirect calls or use system calls directly. Eschweiler et al. [14] target multiple architectures and rely on the (static) control flow structure of the procedure for similarity along with a numeric filtering pre-stage for scalability. This reliance on syntactic features leads to sub-optimal accuracy and limitations, making it not applicable in some cases (such as procedures with a small number of branches).

Horwitz et al. [33] recognized the importance of statistical reasoning, yet their method is geared towards source-code and employs n-grams, which were shown to be a weak representation for binary similarity tasks [11]. Jang et al. [18], who also use this basic n-gram decomposition, show a similarity engine, which can be considered complimentary to our hashed search. However, it targets a different scenario of malware detection using features. N-grams are yet again used by Khoo et al. [19], in combination with graphlets. The approach is structural as well, and thus is susceptible to structural changes and further suffers from limitations similar to [14].

6. Conclusions

We presented a scalable approach for establishing similarity between stripped binaries (with no debug information). Our approach can establish similarity with high accuracy even when the code has been compiled using different compilers, with different optimization levels, or targeted different architectures.

The main idea is to decompose each binary procedure to strands (small comparable segments that preserve dependence chains), and to perform out-of-context "strand re-optimization", by using the compiler-optimizer to transform the strand into a canonical normalized form. After this transformation, these strands can be used for efficient comparison simply by hashing them. To establish procedure similarity, we use a statistical framework that focuses on comparison of *statistically significant* strands, using a global context built using a sampling of procedures "from the wild".

We implemented our technique in a tool called `GitZ` and performed an extensive evaluation. We show that `GitZ` is able to perform millions of comparisons efficiently, and that the combined use of strand re-optimization with the global context establishes similarity with high accuracy, and with an order of magnitude speed-up over a competing method.

Acknowledgment

We would like to thank the developers of `angr.io`, which was instrumental in our research. We would also like to thank Hans Boehm, our shepherd, for helping us improve and clarify the paper.

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.

References

[1] Esh - statistical similarity of binaries. http://binsim.com.

[2] gcc optimizations options. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[3] Llvm's analysis and transform passes. http://llvm.org/docs/Passes.html.

[4] Mcsema. https://github.com/trailofbits/mcsema.

[5] Shellshock vulnerability cve information. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271.

[6] Yard - yet another roc drawer. http://github.com/ntamas/yard.

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

[8] zynamics bindiff manual - understanding bindiff. http://www.zynamics.com/bindiff/manual/index.html#chapUnderstanding.

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

[10] Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, 2016.

[11] Y. David and E. Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 349–360, New York, NY, USA, 2014. ACM.

[12] J. Duan and J. Regehr. Correctness proofs for device drivers in embedded systems. In *5th International Workshop on Systems Software Verification, SSV'10, Vancouver, BC, Canada, October 6-7, 2010*, 2010.

[13] M. Egele, M. Woo, P. Chapman, and D. Brumley. 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.*, pages 303–317, 2014.

[14] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[15] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 480–491, 2016.

[16] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth. 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*, pages 191–201, 2013.

[17] E. R. Jacobson, N. Rosenblum, and B. P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 1–8, New York, NY, USA, 2011. ACM.

[18] J. Jang, D. Brumley, and S. Venkataraman. BitShred : Feature Hashing Malware for Scalable Triage and Semantic Analysis. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 309–320, 2011.

[19] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 329–338, Piscataway, NJ, USA, 2013. IEEE Press.

[20] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*, pages 712–717, 2012.

[21] K. R. M. Leino. This is boogie 2. http://microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[22] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

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

[24] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 238–258. Springer, 2013.

[25] N. Partush and E. Yahav. 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*, pages 811–828, 2014.

[26] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.

[27] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. 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*, pages 709–724, 2015.

[28] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 406–415, New York, NY, USA, 2014. ACM.

[29] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

[30] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the Toolchain Provenance of Binary Code Categories and Subject Descriptors. *20th International Symposium on Software Testing and Analysis (ISSTA)*, page 11, 2011.

[31] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 391–406, New York, NY, USA, 2013. ACM.

[32] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

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

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

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