Jannik Pewny*, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz

# Cross-architecture bug search in binary executables

DOI 10.1515/itit-2016-0040
Received August 17, 2016; accepted November 16, 2016

**Abstract:** With the general availability of closed-source software for various CPU architectures, there is a need to identify security-critical vulnerabilities at the binary level. Unfortunately, existing bug finding methods fall short in that they i) require source code, ii) only work on a single architecture (typically x86), or iii) rely on dynamic analysis, which is difficult for embedded devices. In this paper, we propose a system to derive *bug signatures* for known bugs. First, we compute *semantic hashes* for the basic blocks of the binary. When can then use these semantics to find code parts in the binary that behave similarly to the bug signature, effectively revealing code parts that contain the bug. As a result, we can find vulnerabilities, e.g., the famous *Heartbleed* vulnerabilities, in buggy binary code for any of the supported architectures (currently, ARM, MIPS and x86).

**Keywords:** Binary, bug search, cross-architecture, sampling, similarity metric.

**ACM CCS:** Security and privacy → Software and application security, Security and privacy → Systems security → Vulnerability management → Vulnerability scanners

## 1 Introduction

Software bugs still constitute one of the largest security threats today. Critical software vulnerabilities such as memory corruptions remain prevalent in both open-source and closed-source software [27].

The problem of finding bugs at the source code level has been addressed by a lot of researchers [9, 11, 12, 16, 28]. Professional code verification tools ensure source code quality and a number of automated bug finding proposals analyze source code to find security-critical bugs. However, a lot of prominent software is available only as a binary, either as commercial software (e.g., `MS Office`) or as freely-available closed-source software (e.g., `Adobe Reader` or `Flash`). Software on embedded devices (*firmware*), is usually closed-source, implemented in an unsafe language, and re-uses potentially vulnerable code from third-party projects [5, 26].

Another challenge in finding bugs at the binary level is that more and more software is cross-compiled for various CPU architectures, e.g., when hardware vendors use the same code base to compile firmware for different devices (e.g., home routers, cameras, VoIP phones). Similarly, prominent software such as `MS Office`, `Adobe Reader` or `Flash`, is already available for multiple platforms and architectures, most recently with the increase of ARM-based `Windows RT` deployments. Binaries from varying architectures differ, e.g., in instruction sets, function offsets and function calling conventions. The problem is compounded if cross-compiled software includes well-known, but vulnerable libraries. E.g., after discovery of the *Heartbleed* bug in `OpenSSL`, there is a growing list of affected closed-source software for various architectures (x86, MIPS, ARM, PowerPC, etc.).

In this paper, we make the first step towards finding vulnerabilities *in binary software* and *across multiple architectures*. Once a bug is known—in any binary software compiled to a supported architecture—we aim to identify equally vulnerable parts in other binaries, which were possibly compiled for other architectures. That is, we use a *bug signature* that spans a vulnerable function (or parts of it) to find similar bug instances. While this limits our work to re-finding similar, previously documented instances of bug classes, exactly this problem has evolved to a daily use case in the era of cross-compiled code. Our goal is to assist a human analyst in such scenarios, where the analyst defines a bug signature *once* and then searches for parts in other software binaries—from and to any architecture.

To this end, we propose a mechanism that uses a unified representation of binaries so that we can compare binaries across different architectures. That is, we first lift binary code for any architecture—we currently support In-

***Corresponding author: Jannik Pewny,** Ruhr-Universität Bochum, Lehrstuhl für Systemsicherheit, Universitätsstraße 120, 44780 Bochum, Germany, e-mail: jannik.pewny@rub.de
**Behrad Garmany, Robert Gawlik, Thorsten Holz:** Ruhr-Universität Bochum, Lehrstuhl für Systemsicherheit, Universitätsstraße 120, 44780 Bochum, Germany
**Christian Rossow:** Universität des Saarlandes, Cluster of Excellence on Multimodal Computing and Interaction, Campus E 9 1, Raum 3.07, 66123 Saarbrücken, Germany

tel x86, ARM and MIPS—to an intermediate representation (IR). Based on this IR code, we aim to grasp the *semantics* of the binary at a basic block level. We build assignment formulas for each basic block, which capture the basic block's behavior in terms of input and output variables. An input variable is any input that influences the output variables, such as CPU registers or memory content. We then sample random input variables to monitor their effects on the output variables. This analysis results in a list of input/output (I/O) pairs per assignment formula, which capture the actual semantics of a basic block. Although the *syntax* of similar code is quite different for the various CPU architectures (even in the intermediate representation), we can use such *semantics* to compare basic blocks across ISAs.

We use the semantic representation to search the bug signature in other arbitrary software that is potentially vulnerable to the bug defined in the signature. The bug signature can be derived automatically from a known vulnerable binary program or from source code, and may simply represent the entire vulnerable function. To preserve performance, we use *MinHash* to find suitable basic block matches. Lastly, once basic block matches have been found, we propose an algorithm that leverages the control flow graph (CFG) to expand our search to the entire bug signature. As output, our system lists functions ordered by their similarity to the bug signature.

We systematically test how well our system performs when matching equivalent functions across binaries that have been compiled for different architectures, with different compilers, and using varying optimization levels. The evaluation shows that our system is accurate in matching functions with only a few false positives. We also evaluate our system in various real-world use cases, e.g., our system finds the *Heartbleed* bug in 21 out of 24 tested combinations of software programs across the three supported architectures.

In this work we focus on identifying unpatched bug duplicates, but we envision more use cases of our system, such as binary diffing, searching for software copyright infringement, or revealing code sharing across binaries.

In summary, our four main contributions are as follows:

- We lift ARM, x86 and MIPS code to unified RISC-like expressions that capture I/O syntax per basic block.
- We use a sampling and MinHashing engine to create compact and cheaply-comparable semantic summaries of basic blocks.
- We define a metric to compare code structures like sub-CFGs and functions, which enables us to search for bug signatures in arbitrary software binaries.
- We empirically demonstrate the viability of our approach for multiple real-world vulnerabilities spanning software across three supported architectures.

## 2 Approach

We now outline our general approach for cross-architecture bug search in binary executables. We assume that similar code often stems from slightly modified shared code, which is therefore likely to share the same—possibly security critical—bug.

### 2.1 Workflow

We use a *bug signature*, i.e., a piece of binary code that resembles a specific instance of a vulnerability class, to find possible vulnerabilities in another binary program (*target program*). To this end, we first derive a bug signature

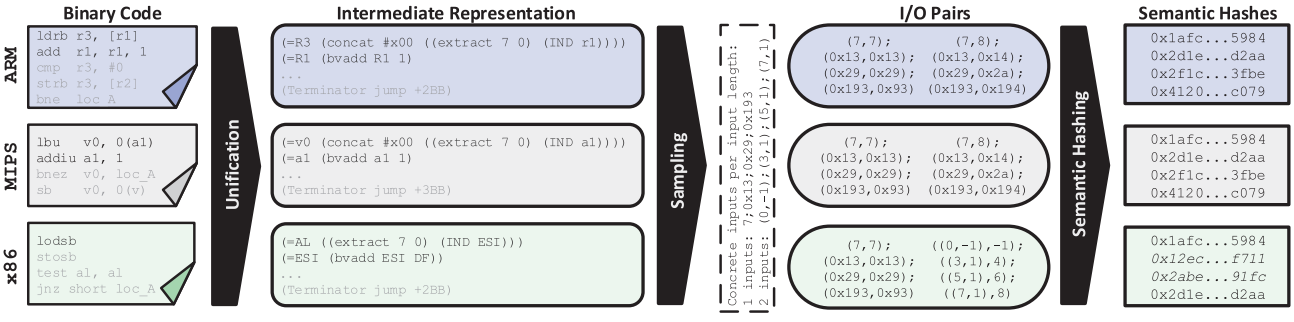



**Figure 1:** Transformation phase: Three binaries (ARM, MIPS, x86) are first unified from assembly to the intermediate representation (S-Expressions), as illustrated for the first instruction (`ldrb`, `lbu` and `lodsb`). Then, using the same random concrete inputs among all formulas, we determine the input/output behavior of the expressions, resulting in a set of I/O pairs. In the last step, we create semantic hashes over these I/O pairs in order to efficiently compare basic blocks.

(Section 2.2). Figure 1 illustrates how we transform the assembly code for both the bug signature and the target program into an intermediate representation (Section 2.3), which results in a list of *assignment formulas*. The assignment formulas detail how an output variable is influenced by its inputs. Next, using random concrete input values (dashed box), we sample the input/output behavior of these formulas (we illustrate sampling of the first formula only). In the last step, we build compact basic block-wise semantic hashes over the I/O pairs, which allow us to efficiently compare the I/O behavior of basic blocks (Section 2.4). All these transformations are one-time processes.

In the search phase, we use the transformed bug signature to identify bugs in the similarly-transformed binaries. That is, we look for promising matching candidates for all individual basic blocks of the bug signature in the target program (Section 2.5). For each such candidate pair, we apply a CFG-driven, greedy, but locally-optimal broadening algorithm. The algorithm expands the initial match with additional basic blocks from the bug signature as well as the target program (Section 2.6) and then computes the similarity between bug signature and target programs, returning a list of code locations ordered by their similarity to the signature.

## 2.2 Bug signatures

A bug signature is just like normal binary code: It consists of basic blocks and possible control-flow transitions between these basic blocks. Therefore, any selection of basic blocks can be used as a bug signature. For example, the bug signature could represent an entire buggy function, minimizing the manual effort. However, bug signatures should be refined to smaller code parts, which cover only the bug and its relevant context. While our target is to search bugs in binaries (i.e., without access to source code), one could use additional aids, e.g. debug information, to automatically find the basic blocks that correspond to the vulnerable function part, effectively deriving bug signatures with almost no manual effort. However, it is quite hard to estimate how the signature size influences results in general: An additional non-characteristic basic block, which is essential to the structure of the vulnerability and discriminative in its context, will likely improve results, while a characteristic basic block, which is non-essential to the vulnerability, may lead to false positives.

## 2.3 Unifying cross-architecture instruction sets

Obviously, the instruction sets of architectures like x86, ARM, and MIPS are quite distinct. Not only the calling conventions and memory accesses (e.g., load/store), but also the general assembly complexity/length, deviate between the architectures. Thus, a key step in our approach is to unify the binary code of different architectures. We first utilize an off-the-shelf disassembler to extract the structure of the binary code (such as functions, basic blocks, and control flow graphs). Then, we transform the complex instructions into simple, RISC-like and unified instructions as it abstracts from architecture-specific artifacts and facilitates symbolic normalization.

## 2.4 Extracting semantics via sampling

The unified instruction set would allow us to compare individual binary instructions syntactically. In the next step, we aim to extract the semantics of the binary code. We aggregate the computational steps for each output variable of a basic block, which gives us precise *assignment formulas* for each output register or output memory location (see Figure 1, second column). For normalization purposes we simplify the assignment formulas by passing them through a theorem prover. We extract the formulas per basic block and we found this to work quite well in scenarios with different architectures, different compilers, or different optimization levels.

At this point, we have precise descriptions of the effect of a basic block on the programs state. However, these descriptions are still purely syntactic. To achieve our goal of bug finding, we relax the condition to find *code equivalence* and use a metric that measures *code similarity*. We build such a metric upon *sampling*, which was proposed by Jin et. al [13]. First, we generate random and concrete values, which we use as inputs for the formulas of each basic block. Such point-wise evaluations capture the semantics of the performed computation. Now, similarity can be expressed through the assumption that similar pieces of code have more input/output pairs in common than dissimilar ones.

## 2.5 Similarity metric via semantic hashes

The higher the number of shared, equal I/O pairs between two basic blocks, the higher is the similarity of these basic blocks. However, comparing this many I/O pairs would be

computationally expensive. We tackle this bottleneck with locally-sensitive hashes. A LSH has the property that the similarity of the hash reflects the similarity of the hashed object. We chose to use MinHash [3], which satisfies the LSH properties and converges against the Jaccard index. Comparing two basic blocks now only requires comparing their semantic hashes.

### 2.6 Comparing larger binary structures

At this point, we can find pairs of similar basic blocks, which are candidates for a signature-spanning match. In order to match an entire bug signature, though, comparing individual basic blocks is not sufficient.

First, we pick a basic block from the bug signature and compare it to all basic blocks from the program in question. Then, we use an algorithm called *Best Hit Broadening* (BHB), which broadens the initial candidate match along the CFGs in the bug signature and target program. BHB operates in a greedy, but locally-optimal manner, until the match spans the entire signature. BHB is then repeated for all basic blocks in the bug signature, resulting in a list of functions ordered by their similarity to the signature (see Section 3.4 for details).

## 3 Implementation

In this section, we discuss some of the specific details for the proof-of-concept implementation of our approach.

### 3.1 Common ground: The IR stage

First, we use `IDA Pro` [10] to extract a disassembly and the control-flow graph from the binary. To find a common representation for binary code we utilize the VEX-IR, which is the RISC-like intermediate representation for the popular `Valgrind` toolkit. We leveraged `pyvex` [25], a Python framework with bindings to the VEX-IR, and used its API to process the IR statically. Afterwards, we use the `Z3` theorem prover [19] to simplify and normalize expressions.

### 3.2 Extracting semantics: Sampling

To grasp the semantic I/O behavior of a basic block, we evaluate its formulas point-wise, by using random vectors with elements from the range $[-1000, 1000]$ as input. We then create unique I/O pair representations by computing a 64-bit CRC checksum of the input length, the inputs, and the output value.

To ensure that our sampling is robust to the order of variables, we also use all permutations of the concrete value vectors as inputs. Regardless of the CPU architecture, almost all basic blocks' formulas have only a few input variables, so we can safely limit sampling to those formulas with at most four variables.

### 3.3 Semantic hash

MinHash applies a (non-cryptographic) hash function to each element in the list and stores the minimal hash value among all I/O pairs. The hash function randomly selects an estimator for set-similarity and similarity is expressed as the average over $i = 0..n$ such estimators: $sim(mh_1, mh_2) := |\{mh_1[i] = mh_2[i]\}|/n$.

For our evaluation, we used an affine hash function of the form $h(x) := ax + b \bmod p$ with random 64-bit coefficients, where a prime modulus $p$ guarantees that all coefficients are generators. To improve performance, we simulate further hash functions by transforming the output of the real hash function with rotation and XORing ($t(h(x)) := rotate(h(x), c) \oplus d$), which changes the order of elements and therefore the selected minimum, which suffices for MinHashing. We use 800 such hash functions, which leads to an error of about 3.5% [29].

We implemented two improvements of the standard MinHash algorithm. First, we compute multiple MinHashes per basic block, which we denote as *Multi-MinHash*. We do so by splitting the formulas into groups according to the number of input variables per formula and computing one MinHash per group. Thus, to compare two basic blocks, we compute $\frac{\sum_i s_i \cdot (w_i + w_i')}{\sum_i (w_i + w_i')}$, where $s_i$ is the similarity of the formulas with $i$ variables, $w_i$ and $w_i'$ the number of formulas with that number of variables in the respective basic block. This solves the issue of under-represention in the hash value for formulas with only a few samples (e.g., very few or no inputs). Second, we do not only store the smallest hash value per hash function, but the $k$ smallest hash values—which we denote as *k-MinHash* (we use $k = 3$).

### 3.4 Bug signature matching

```
BHB(sp_i ∈ SP, sp_t ∈ C_i, Sig)
  M_S = [], M_T = []
  PQ = (sim(sp_i, sp_t), (sp_i, sp_t))
  while(PQ ≠ ∅)
    P_S, P_T := PQ.pop()
    M_S += P_S, M_T += P_T

    D_p := sim_mat(P_S.pred \ M_S, P_T.pred \ M_T)
    M̃_p := Hungarian(D_p)
    PQ.add(sim(P_i, P_j), (P_i, P_j)), ∀(P_i, P_j) ∈ M̃_p

    D_s := sim_mat(P_S.succ \ M_S, P_T.succ \ M_T)
    M̃_s := Hungarian(D_s)
    PQ.add(sim(P_i, P_j), (P_i, P_j)), ∀(P_i, P_j) ∈ M̃_s

  dist := Σ_{i=0}^{|M_S|-1} sim(M_S[i], M_T[i])
  return dist / |Sig|
```

**Listing 1.** Best-Hit-Broadening Algorithm.

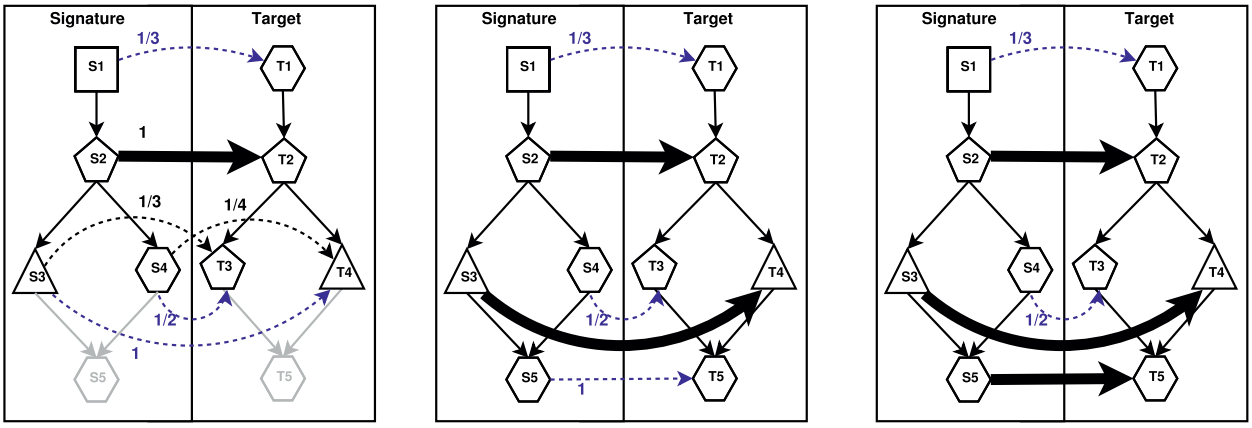
BHB (Listing 1) works as follows: Given a pair of starting points (a basic block from the signature and its corresponding matching candidate in the target program), it first explores the immediate neighborhood of these basic blocks along their respective CFGs (Figure 2a), strictly separating forward and backward directions. After finding a locally-optimal matching (using the *Hungarian method* [7]) among the newly discovered neighborhood nodes, it picks the basic block pair with the maximum similarity (Figure 2b). The broadening is greedy, avoiding expensive backtracking steps. This process is repeated (Figure 2c) until all basic blocks from the bug signature have been matched.

## 4 Evaluation

The experiments were conducted on an Intel Core i7-2640M @ 2.8 GHz with 8 GB DDR3-RAM and included 60 binaries from three architectures (x86, ARM, MIPS; all 32 bit) and three compiler versions (`gcc` v4.6.2/v4.8.1 and `clang` v3.0). While we focussed on Linux, we encountered different core libraries, especially for router software (`DD-WRT`/`NetGear`/`SerComm`/`MikroTik`). We also compared binaries across Windows and Mac OS X, i.e., covered all three major OSes.

### 4.1 False/true positives across compilers/code optimization

Next, we systematically evaluated how the choice of compiler and optimization level influence the accuracy of our algorithm. We chose the largest `coreutils` programs and compared all 12 binaries with each other (i.e., all 144 binary pairs) using function-wise matching. Figure 3 illustrates the results of this experiment in a matrix.




(a) First BHB round with the initial starting point and its candidate match (annotated with the bold arrow).

(b) After the first BHB round, another pair of BBs is matched and the lower two nodes are now adjacent.

(c) After the third step, three pairs are matched. There are no further neighbors and the other two matches are trivial.

**Figure 2:** BHB example in three steps: bug signature on the left, target program on the right side. The difference between two basic blocks is visualized by the different numbers of edges ($n$) of the shapes. The similarity is then calculated as $\frac{1}{1+n}$.

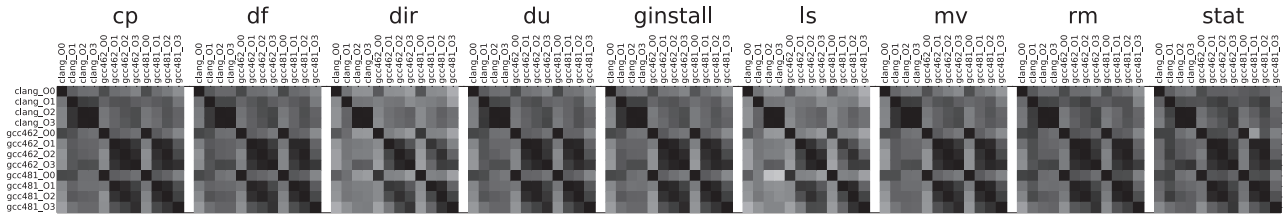



**Figure 3:** True positive matrix for the largest `coreutils` programs, compiled with three different compilers and four different optimization levels. Darker colors represent a higher percentage of correctly matched functions.

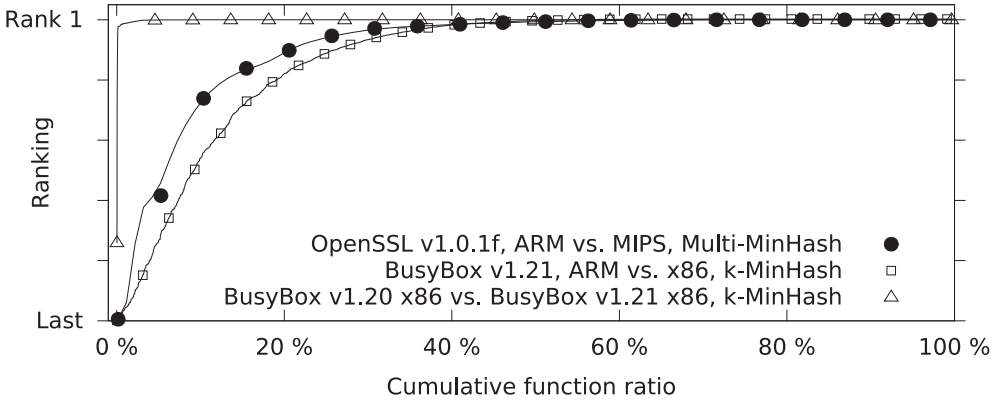



**Figure 4:** Ranks of true positives in function matching for `OpenSSL` and `BusyBox`. Other architecture-combinations are similar and were omitted.

## 4.2 False/true positives across architectures

In this experiment, we compared all functions of one binary $A$ to all functions of another binary $B$. Ideally, every function would appear at rank #1 in the function ranking. Figure 4 shows a cumulative distribution function (CDF) on the ranks of all functions. Our results are close to the ideal for the same architecture and provide a sufficient degree of similarity across architectures in most cases.

## 4.3 Bug search in closed-source software

In this section, we evaluate several case studies of recent prominent cross-architectural vulnerabilities. For example, we find the two functions vulnerable to the *Heartbleed* bug, `tls1_process_heartbeat` (TLS) and `dtls1_process_heartbeat` (DTLS), on desktops as well as on mobile devices and router firmware images (see Table 1). Similarly, we found `BusyBox` bugs, which are part of closed-source bundles of embedded devices (e.g., home routers) across multiple architectures, and identify both a bug and a backdoor in closed-source firmware images.

Furthermore, we analyzed a vulnerability (CVE-2013-6484) in `libpurple`, which is used in `Pidgin`, a popular instant messenger for Windows and Linux, with a `Mac OS X` counterpart, `Adium`. The vulnerable function in `Pidgin` contained many basic blocks of other inlined functions, whereas `Adium` did not inline them. Full-function matching, which is used in most modern bug-finding techniques, cannot produce good results in such cases. However, when choosing only those basic blocks from the function start through the vulnerability, while avoiding the early-return error states, we achieved rank #1 in both cases.

**Table 1:** Ranks of functions vulnerable to *Heartbleed* in `OpenSSL` compiled for ARM, MIPS and x86, in `ReadyNAS` v6.1.6 (ARM) and `DD-WRT` r21676 (MIPS) firmware. Each cell gives the ranking of the TLS/DTLS function.

| | | | Multi-MH | | Multi-k-MH | |
|---|---|---|---|---|---|---|
| From | → | To | TLS | DTLS | TLS | DTLS |
| ARM | → | MIPS | 1;2 | 1;2 | **1;2** | 1;2 |
| ARM | → | x86 | 1;2 | 1;2 | 1;2 | 1;2 |
| ARM | → | DD-WRT | 1;2 | 1;2 | 1;2 | 1;2 |
| ARM | → | ReadyNAS | 1;2 | 1;2 | 1;2 | 1;2 |
| MIPS | → | ARM | 2;3 | 3;4 | 1;2 | 1;2 |
| MIPS | → | x86 | 1;4 | 1;3 | 1;2 | 1;3 |
| MIPS | → | DD-WRT | 1;2 | 1;2 | 1;2 | 1;2 |
| MIPS | → | ReadyNAS | 2;4 | 6;16 | 1;2 | 1;4 |
| x86 | → | ARM | 1;2 | 1;2 | 1;2 | 1;2 |
| x86 | → | MIPS | 1;7 | 11;21 | 1;2 | 1;6 |
| x86 | → | DD-WRT | 70;78 | 1;2 | 5;33 | 1;2 |
| x86 | → | ReadyNAS | 1;2 | 1;2 | 1;2 | 1;2 |

**Table 2:** Runtime in minutes in the offline-phase.

| Entity | #BBs | IR-Gen. | Multi-MH | k-MH |
|---|---|---|---|---|
| BusyBox, ARM | 60,630 | 28 | 80.3 | 1522 |
| BusyBox, MIPS | 67,236 | 35 | 86.9 | 1911 |
| BusyBox, x86 | 65,835 | 51 | 85.0 | 1855 |
| OpenSSL, ARM | 14,803 | 8 | 16.3 | 349.7 |
| OpenSSL, MIPS | 14,488 | 9 | 16.3 | 434.0 |
| OpenSSL, x86 | 14,838 | 12 | 20.2 | 485.2 |
| **Normalized Avg.** | 10,000 | 6.2 | 12.5 | 279.9 |

### 4.4 Unpatched vs. patched code

Naturally, a patched version is very similar to an unpatched one which causes false positives. However, one could create two bug signatures, one for the vulnerable function and one for the patched function. E.g., for the bugs in OpenSSL and BusyBox, either bug signature (patched or unpatched) ranked both software versions at rank #1, where the unpatched version is more similar to the "buggy" bug signature (considerably in three cases, marginally in one). This could give valuable information about whether one is looking at a false positive.

### 4.5 Performance

Table 2 shows the average timings achieved in real-life programs, showing that IR-formulas can be generated in a reasonable time for all programs under analysis. It further shows that sampling and MinHashing of real-life programs scales. Typically, the runtime for the signature search in various scenarios is in the order of minutes for our real-world use cases.

## 5 Discussion

One challenge that we already touched upon in Section 4 is the fact that our approach cannot verify that the code part that was found is actually vulnerable. Such an automatic verification would be ideal, but is outside the scope of this work.

Comparing software at the binary level is challenging, since its representation (even on a single architecture) heavily depends on the exact build environment. However, many compiler optimization techniques like register spilling, instruction reordering, common subexpression elimination, constant folding and different calling conventions, are already handled by our approach.

Our approach is rather slow, but most computations can be scaled up with straightforward parallelization. In addition, note that the most compute-intensive parts of our approach are one-time operations.

## 6 Related work

To the best of our knowledge, we are the first to propose a strategy for comparing binary code across different architectures, which makes it hard to compare it to other works that operate under less challenging conditions (e.g., available source code, a single architecture, or function-granularity).

A first line of research has proposed methods to find code clones on a source code level. Examples are CCFinder [16], DECKARD [12], and ReDeBug [11]. Yamaguchi et al. [28] are the first to describe the extrapolation of vulnerabilities by finding close neighbors. They, as well as Gauthier et al. [9], further describe finding missing checks through anomaly detection. Modern approaches for binaries follow strategies to compare the semantics of code. Examples are TEDEM [23], Exposé [22], BinHunt [8], its follow-up project, iBinHunt [21], and BinJuice [17], which uses syntactic equations similar to our formulas and hashes those to measure similarity. BinHash [14] inspired our sampling, which is also used by *Blanket Execution* (Blex) [6]). Both, however, do not include steps to support multiple architectures or compare sub-function granular code constructs.

Orthogonal related work searches for previously-unknown bugs. Examples are Coverity [2] AEG [1], or Mayhem [4]. Other systems use type-inference [15, 24], Fuzzing [20] or Tainting [18].

## 7 Conclusions

We showed that semantic binary code matching is possible across CPU architectures under realistic assumptions. This advances prior research results that are restricted to comparing binary code of a single architecture. Our novel metric allows for fine-grained code comparison, which we successfully applied to identify real-world vulnerabilities in closed-source software.

**Acknowledgement:** This work was supported by ERC Starting Grant No. 640110 (BASTION) and German Research Foundation (DFG) research training group UbiCrypt (GRK 1817).

## References

1. T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Symposium on Network and Distributed System Security (NDSS)*, 2011.
2. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of ACM*, 2010.
3. A. Z. Broder. On the Resemblance and Containment of Documents. In *IEEE SEQUENCES*, 1997.
4. S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy*, 2012.
5. A. Costini, J. Zaddach, A. Francillon, and D. Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security Symposium*, 2014.
6. M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security Symposium*, 2014.
7. A. Frank. On Kuhn's Hungarian Method – A Tribute From Hungary. Technical report, Oct. 2004.
8. D. Gao, M. Reiter, and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Information and Comm. Sec.*, 2008.
9. F. Gauthier, T. Lavoie, and E. Merlo. Uncovering Access Control Weaknesses and Flaws with Security-discordant Software Clones. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.
10. IDA Pro – Interactive Disassembler. http://www.hex-rays.com/idapro/.
11. J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *IEEE Symposium on Security and Privacy*, 2012.
12. L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *International Conference on Software Engineering (ICSE)*, 2007.
13. W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary Function Clustering Using Semantic Hashes. In *ICMLA*, 2012.
14. W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary function clustering using semantic hashes. In *ICMLA*, 2012.
15. R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs with Type Inference. In *USENIX Security Symposium*, 2004.
16. T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.*, 2002.
17. A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast Location of Similar Code Fragments Using Semantic 'Juice'. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2013.
18. V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, 2005.
19. Microsoft-Research. Z3: Theorem Prover, February 2014. http://z3.codeplex.com/.
20. B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commununications of ACM*, 1990.
21. J. Ming, M. Pan, and D. Gao. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *International Conference on Information Security and Cryptology*, 2013.
22. B. H. Ng and A. Prakash. Expose: Discovering Potential Binary Code Re-use. In *IEEE COMPSAC*, 2013.
23. J. Pewny, F. Schuster, C. Rossow, and T. Holz. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.
24. U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format-String Vulnerabilities with Type Qualifiers. In *USENIX Security Symposium*, 2001.
25. Y. Shoshitaishvili. Python bindings for Valgrind's VEX IR, February 2014. https://github.com/zardus/pyvex.
26. UBM Tech. Embedded Market Study, Mar. 2013. http://tinyurl.com/embmarketstudy13.
27. V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory Errors: The Past, the Present, and the Future. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2012.
28. F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.
29. R. B. Zadeh and A. Goel. Dimension Independent Similarity Computation. *CoRR*, abs/1206.2082, 2012.

## Bionotes

**Jannik Pewny, M.Sc.**
Ruhr-Universität Bochum, Lehrstuhl für Systemsicherheit,
Universitätsstraße 120, 44780 Bochum, Germany
**jannik.pewny@rub.de**

Jannik Pewny, M.Sc., studied IT-Security and Applied Informatics at the Ruhr-University Bochum, Germany (M.Sc. 2012, B.Sc. 2013). Since then, he worked for Prof. Dr. Thorsten Holz at the chair for System Security, where his dissertation topic is Retrofitting Security into Legacy Software Systems. In short, he tries to introduce the merits of modern IT-Security into old software.

**Dipl. Inform. Behrad Garmany**
Ruhr-Universität Bochum, Lehrstuhl für Systemsicherheit,
Universitätsstraße 120, 44780 Bochum, Germany
**behrad.garmany@rub.de**

Dipl. Inform. Behrad Garmany studied Informatics at the RWTH Aachen, Germany (Dipl. Inform. 2012). He also works at the same chair for System Security for Prof. Dr. Thorsten Holz, studying modern attack and defense vectors against modern software. Lately, he focussed his efforts towards bounded model checking.

**Dipl. Biol. Robert Gawlik**
Ruhr-Universität Bochum, Lehrstuhl für Systemsicherheit,
Universitätsstraße 120, 44780 Bochum, Germany
**robert.gawlik@rub.de**

Dipl. Biol. Robert Gawlik developed a taste for exploitation and bug hunting during his original career path, the study of biology. He switched his field of work and is now a colleague of the former two authors at the chair for System Security in Bochum. He still focussed mainly on modern attack vectors, where he often targets web browsers.

**Dr. Christian Rossow**
Universität des Saarlandes, Cluster of Excellence on Multimodal Computing and Interaction,Campus E 9 1, Raum 3.07,
66123 Saarbrücken, Germany
**crossow@mmci.uni-saarland.de**

Dr. Christian Rossow obtained his PhD degree from the VU Amsterdam in April 2013, worked at the Institute for Internet Security if(is) in Gelsenkirchen, Germany, and was a postdoctoral researcher at the VU Amsterdam (Herbert Bos) and the Ruhr University Bochum (Thorsten Holz). Since June 2014, he leads the System Security research group at the Saarland University in Germany since June 2014.

**Prof. Dr. Thorsten Holz**
Ruhr-Universität Bochum, Lehrstuhl für Systemsicherheit,
Universitätsstraße 120, 44780 Bochum, Germany
**thorsten.holz@rub.de**

Prof. Dr. Thorsten Holz is Professor at the faculty for electrical engineering and information technology at the Ruhr University Bochum, Germany. His research focus is on applied aspects of security IT-systems and machine-oriented IT-security. He earned his degree in informatics at the RWTH Aachen, Germany in 2005 and his PhD at the University Mannheim, Germany in 2009. Before he his call to the Ruhr-University, he worked as a Post-Doc at the institute of Computer-Aided Automation at the Technical University in Vienna. In 2011, he received the Heinz-Maier-Leibnitz Prize.