QuickBCC: Quick and Scalable Binary Vulnerable Code Clone Detection

Hajin Jang, Kyeongseok Yang, Geonwoo Lee, Yoonjong Na, Jeremy D. Seideman, Shoufu Luo, Heejo Lee, Sven Dietrich

To cite this version:

Hajin Jang, Kyeongseok Yang, Geonwoo Lee, Yoonjong Na, Jeremy D. Seideman, et al.. QuickBCC: Quick and Scalable Binary Vulnerable Code Clone Detection. 36th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC), Jun 2021, Oslo, Norway. pp.66-82, 10.1007/978-3-030-78120-0_5 . hal-03746026

HAL Id: hal-03746026

https://inria.hal.science/hal-03746026

Submitted on 4 Aug 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

QuickBCC: Quick and Scalable Binary Vulnerable Code Clone Detection

Hajin Jang[1†], Kyeongseok Yang[1†], Geonwoo Lee[1†], Yoonjong Na[1], Jeremy D. Seideman[2], Shoufu Luo[2], Heejo Lee[1*], and Sven Dietrich[2*]

[1] Korea University, Seoul, South Korea
{hajin_jang,ks8171235,2016320146,nooryyaa,heejo}@korea.ac.kr
[2] City University of New York, New York, USA
{jseideman,sluo2}@gradcenter.cuny.edu, spock@ieee.org

Abstract. Due to code reuse among software packages, vulnerabilities can propagate from one software package to another. Current code clone detection techniques are useful for preventing and managing such vulnerability propagation. When the source code for a software package is not available, such as when working with proprietary or custom software distributions, binary code clone detection can be used to examine software for flaws. However, existing binary code clone detectors have scalability issues, or are limited in their accurate detection of *vulnerable code clones*.

In this paper, we introduce QuickBCC, a scalable binary code clone detection framework designed for vulnerability scanning. The framework was built on the idea of extracting semantics from vulnerable binaries both before and after security patches, and comparing them to target binaries. In order to improve performance, we created a signature based on the changes between the pre- and post-patched binaries, and implemented a filtering process when comparing the signatures to the target binaries. In addition, we leverage the smallest semantic unit, a *strand*, to improve accuracy and robustness against compile environments. QuickBCC is highly optimized, capable of preprocessing 5,439 target binaries within 111 minutes, and is able to match those binaries against 6 signatures in 23 seconds when running as a multi-threaded application. QuickBCC takes, on average, 3ms to match one target binary. Comparing performance to other approaches, we found that it outperformed other approaches in terms of performance when detecting well known vulnerabilities with acceptable level of accuracy.

Keywords: Binary Code Clone · Static Analysis · Security Vulnerability · Patch Signature

[†] Hajin Jang, Kyeongseok Yang, and Geonwoo Lee contributed equally to this work.
[*] To whom all correspondence should be addressed.

1 Introduction

Vulnerability propagation, as a result of code reuse, often occurs during the software development life-cycle. One of the best methods to mitigate vulnerability propagation is through *code clone detection*. Tools that perform code clone detection accomplish this by searching for duplicated code fragments in a source code base [18]. When leveraged to find vulnerable code clones, these tools can assist in finding propagated vulnerabilities which originated in external software components. The latest source code clone detector has already been effectively used in practice [19].

Binary code-level vulnerable code clone detection is as important as source code-level vulnerable code clone detection, for three main reasons. First, many programs and firmware are only distributed as binaries, such as commercial operating systems and proprietary software. Second, a user often downloads and runs many open-source software (OSS) programs as a precompiled binary. Vulnerabilities can be added or removed while software distributors compile original source code using their compile options. Finally, a vulnerability can also propagate through *library linking*. Even if the source code of a program itself is free from vulnerabilities, one of its dependent libraries can introduce a vulnerability, regardless of how secure the program itself is.

However, existing binary code clone detectors have issues with scalability. Additionally, the environment in which the program was compiled can dramatically affect the structure of the binary, and it is one of the main obstacles of the binary code clone detection [24]. Some detectors focus on accuracy by relying on more precise, but slower techniques. While the satisfiability modulo theories (SMT) solver or symbolic execution with a theorem prover can cover up the change caused by the compile environment, they also slow down the overall process [13, 15].

Another issue is that many existing binary code clone detectors can compute the similarity of two target functions, but cannot efficiently distinguish between a vulnerable binary and a patched binary. For example, when there is a vulnerable function A and a patched function B, the fundamental structures of these two functions are the same. Therefore, a study that only performs code clone detection using a simple similarity metric tends to consider A and B to be similar.

For these reasons, we introduce QuickBCC, a scalable binary code clone detection framework for vulnerability scanning. We built QuickBCC to find vulnerable code clones while achieving fast performance with a high degree of accuracy. It operates by extracting semantics from binaries before and after a security patch and statically comparing them to target binaries, balancing performance and accuracy. In order to improve the performance, the signature was built from the changes between the pre- and post-patched binaries, and a filtering process was put in place before comparing it to the target binaries. For example, rather than comparing all parts of the target function with all parts of the signature function, comparing them with only the changed parts of the signature function (Fig. 1) can be done much faster. In addition, we improved accuracy and ro-

```
1   ...
2           goto err;
3           ERR_clear_error();
4       }
5   } else {
6   /* Only exit on fatal errors*/
7   if (pkcs7_decrypt_rinfo(&ek, ...
8    goto err;
9    ERR_clear_error();
10   }
11   ...
```

(a) Target Binary

```
1   ...
2           goto err;
3           ERR_clear_error();
4       }
5   } else {
6   /* Only exit on fatal errors*/
7   if (pkcs7_decrypt_rinfo(&ek, ...
8    goto err;
9    ERR_clear_error();
10   }
11   ...
```

(b) Whole code as a signature

```
...
        goto err;
        ERR_clear_error();
    }
} else {
/* Only exit on fatal errors, not decrypt failure *
− if (pkcs7_decrypt_rinfo(&ek, &eklen, ri, pkey) <
+ if (pkcs7_decrypt_rinfo(&ek, &eklen, ri, pkey, 0
 goto err;
 ERR_clear_error();
 }
 ...
```

(c) Partial code as a signature

Fig. 1: Changed code comparison

bustness against compile environments by lifting binary code to an intermediate representation (IR) and extracting the smallest semantic unit, the *strand* [9]. Because a strand provides the smallest data flow, it can respond to a variety of environments.

We accelerated the overall detection process by selecting and extracting only the core data flow (semantics that trigger the vulnerability) from a given vulnerability, and heavily optimizing the framework. We have seen that QuickBCC took an average of only 3ms to match a target binary. It can preprocess 5,439 target binaries within 111 minutes while matching them against 6 vulnerability signatures only takes 23 seconds when multi-threaded. QuickBCC proved its accuracy by detecting one signature from multiple binaries. For example, QuickBCC detected CVE-2019-1547 and CVE-2019-1563 from every vulnerable version of OpenSSL binaries included in the target set. Even if the framework cannot overcome every difference presented by different compile environments, its high performance can be exploited to widen its coverage.

We summarize our contributions as follows:

- **Practical detection of binary code clones:** We present the QuickBCC framework, which is scalable and fast enough to be used as a practical vulnerability detection method.
- **Vulnerable code clone detection:** We propose a novel approach of distinguishing vulnerable code clones from ordinary binary code by introducing the ideas of a *removal mark* and an *addition mark*.

2 Related Works

2.1 Source Code Clone Detection

Source code clone detection aims to find code clones within the program source code; Some detection methods can also be used to detect vulnerabilities present in source code, by comparing program code to known vulnerable code.

For example, the authors of ReDeBug [17] implemented a system to find unpatched code in OS distribution. They used a tokenization technique to normalize source code, and enabled high-performance large-scale analysis of code. VUDDY [19] is a scalable vulnerable code clone detector with function-level granularity. The normalization techniques proposed in that research enabled faster large-scale analysis of code while detecting more types of code clones and reporting significantly low false positives. MVP [26] is also a scalable vulnerable code clone detector, operating slower than the other two approaches, but with higher precision and recall, by using the vulnerability signature of a function and its patch signature.

2.2 Binary Code Clone Detection

Binary code clone detection methods are capable of detecting binary code (i.e. machine code) clones or their corresponding assembly expression. For example, BinSequence [16] was designed to disassemble, normalize, and compare code fragments by creating fingerprints at the instruction level, the basic block level, and the structure level. Their approach works by building up signatures from paths in basic blocks. In contrast, our method works by looking at the sets of instructions in order to take into account differences in compile environments.

The *Esh* system [9] detects similar procedures based on the idea of *similarity by composition*, by which they break code up into smaller fragments called strands and measure similarity between strands of two binary functions. Inspired by *Esh*, our work follows the same idea, building up a function similarity metric from a strand similarity while keeping performance in mind by introducing a light-weight similarity metric with filtering methods.

XMATCH [14], which can be used in cross-platform bug search, lifts raw binary codes to IR, then extracts a conditional formula. Extracted formula is the semantic feature of raw binary code and used to matching similar binaries.

Another approach is to use Control Flow Graphs (CFG) to compare binaries. BinHunt [15] is useful for finding similar programs based on semantics, and can be useful in comparing CFGs. However, this method does not scale well with a large number of differences in the binaries. BinARM [25] compares binaries with fuzzy matching accelerated by extensive filtering with features and execution paths of binary program.

Some research tried to integrate techniques frequently used in machine learning into binary code clone detection. discovRE [13], designed to measure structural similarity of binaries using approximated subgraph isomorphism for bug search, was able to accelerate the overall comparison running time by filtering out irrelevant target functions using a kNN-based filter. Asm2Vec [11] improved robustness of binary search clone detection against compiler optimization and obfuscation, achieving its goal through learning vector representations of an assembly function. Gemini [27] used a deep neural network to generate embeddings of binary function and calculated similarity of binary functions as the distance between two embeddings. SAFE [21] provided architecture to calculate the binary similarity using embedding of fucntions on self-attentive neural network.

INNEREYE [30], and DEEPBINDIFF [12] leveraged NLP techniques to implement a robust and cross-architectural binary code clone detector. INNEREYE interpreted an instruction as a word and a basic block as a sentence to generate expressions of binary functions. DEEPBINDIFF extracted code semantics and control flows of a binary, and used a k-hop greedy matching algorithm to measure the difference between block embeddings. However, a learning-based approach requires extensive training time before detecting binary code clones. In contrast, QuickBCC can be run properly on ordinary desktop computer, without the powerful GPU or specialized hardware often required for efficient neural network computation.

The FIBER system [29] was designed to sit between source and binary code clone detection, by attempting to use as much source code information as possible to help generate a binary signature. Their system was designed to look for patch presence – whether a particular vulnerability has had a patch applied to mitigate it. While the FIBER system leverages source code and patch information to generate accurate signatures, our primary focus is on detecting binary code clones when source code is not available. SPAIN [28] also used patched binary and its corresponding vulnerable binary to extract the patch pattern and later search similar patches using the extracted patch pattern.

Hash-based approaches, such as MinHash [23], capture semantics from binary functions to create a signature, and then use the hashed signature for comparison. Our method uses a hash-based approach in order to define and easily compare our function representation. The general subject of binary signatures and fingerprints has been explored in a recent study [7], leading to research in authorship, code reuse, and binary semantics.

FirmUP [10] analyzes vulnerabilities by extracting strands from all functions of the vulnerable binary and comparing the two target binaries through the *Back-and-Forth Game algorithm*. This method has a large overhead in terms of size and speed; the entire binary of the target vulnerability is used, and all functions within are compared, so execution speed is very slow.

3 Approach

In Section 3, how QuickBCC compute the similarity of two binary function is explained. And overview and detail of QuickBCC's architecture is also covered in this section.

3.1 Similarity and Equivalence metric

The goal of this work is to design a function-level binary code clone detector, $\mathcal{D}$, that detects **semantic clones** [8]. Two functions F, F' are semantic clones if they provide the same functionality, denoted as $F \equiv F'$.

The code clone detector $\mathcal{D}$ consists of two components: a scoring function $\mathcal{B} : \mathbb{F} \times \mathbb{F} \to [0, 1]$, which takes two functions F_x and F_y as its input and computes

a similarity score, and a threshold value t, such that

$$F_x \equiv F_y \;\; if \;\; \mathcal{B}(F_x, F_y) \geq t \; , \quad F_x \not\equiv F_y \;\; if \;\; \mathcal{B}(F_x, F_y) < t \tag{1}$$

In this work, we propose the scoring function $\mathcal{B}$ as a similarity measurement built on *n-gram similarity* [20] of *strands* [9]. First, a function F_x is decomposed into a set of strands, i.e. $F_x = \{s_1, s_2, \cdots\}$. Given a strand $s = i_1 i_2 i_3 \cdots i_k$ where i_k are instructions, we derive a n-gram set s' from s by extracting all n-grams of instructions. For example, if $n = 3$, we can build an n-gram set as:

$$s^{'} = \{i_1 i_2 i_3, i_2 i_3 i_4, i_3 i_4 i_5 \cdots\} \tag{2}$$

In order to quantitatively determine the similarity, we used the Jaccard Index. The Jaccard index is a representative algorithm that obtains the similarity between two sets when the data does not have an order or quantity, is used. The *n-gram similarity* of two strands s_x and s_y is defined as the Jaccard Index of their corresponding n-gram sets, s'_x and s'_y respectively:

$$Sim(s_x, s_y) = \frac{|s^{'}_x \cap s^{'}_y|}{|s^{'}_x \cup s^{'}_y|} \tag{3}$$

If F_x, F_y have n, m strands (e.g. $\{s_{x_1}, s_{x_2}, \cdots, s_{x_n}\}$, $\{s_{y_1}, s_{y_2}, \cdots, s_{y_m}\}$) respectively, then we calculate the similarity score between the two functions as:

$$\mathcal{B} = Score(F_x, F_y) = \frac{1}{n} \sum_{i=1}^{n} \max_{j=[1,m]} (Sim(s_{x_i}, s_{y_j})) \tag{4}$$

To provide symmetry to the scoring function $\mathcal{B}$, $\mathcal{B}$ can be modified as:

$$\mathcal{B} = max(Score(F_x, F_y), Score(F_y, F_x)) \tag{5}$$

Therefore, using $\mathcal{B}$ can quantitatively evaluate the similarity between the two functions. QuickBCC uses $\mathcal{B}$ defined in Equation (5).

Now we present an overview of QuickBCC's architecture in Figure 2. QuickBCC consists of three main steps: (1) vulnerability preprocessing, (2) target binary preprocessing, and (3) matching. Each step makes use of one or more of its major components, the binary preprocessor, and the code clone detector.

3.2 Binary Preprocessor

Disassembler The fingerprint generator uses radare2 [6] to retrieve basic blocks and disassembled functions from binary. The disassembler works by performing a control flow graph (CFG) analysis on each function. When the optional addition and removal marks (Section 3.3) are supplied, the preprocessor works only on functions and instructions indicated by the marks. We used diaphora [4], an open-source binary diffing tool running on IDA [3] to create vulnerability signatures (Section 3.3).

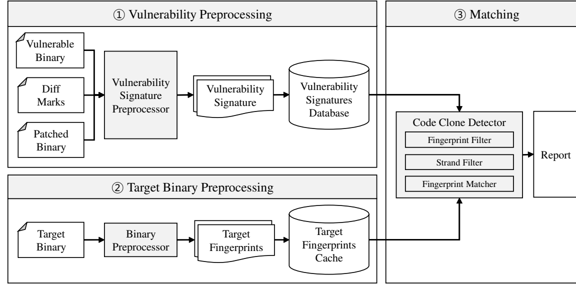

Fig. 2: Overview of the QuickBCC Architecture

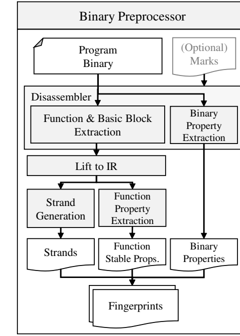

Fig. 3: Overview of the Binary Preprocessor

Lifting to IR QuickBCC leverages on an *intermediate representation* (IR) rather than relying on specific instruction sets. First, the use of an IR allows us to write a much simpler strand generation algorithm, which contributes to the performance. Additionally, we can reuse one codebase and algorithm over multiple architectures. We chose VEX-IR [2, 22] as the basis of our work. VEX-IR is capable of *static single assignment* (SSA), which lets us create strands with a relatively simple algorithm.

```
 1 ¦ t2 = LDle:I32(0x0007bc64)
 2 ¦ PUT(a4) = t2
 3 ¦ t3 = GET:I32(r13)
 4 ¦ t5 = Add32(t3,0x00000014)
 5 ¦ PUT(v1) = t5
 6 ¦ PUT(ip) = 0x0007b620
 7 ¦ t8 = LDle:I32(t2)
 8 ¦ PUT(a1) = t5
 9 ¦ t14 = Add32(t8,0x00000001)
10 ¦ PUT(a3) = t14
11 ¦ PUT(ip) = 0x0007b62c
12 ¦ STle(t2) = t14
13 ¦ PUT(lr) = 0x0007b634
NEXT: PUT(pc) = 0x00027648; Ijk_Call
```

Fig. 4: Example of VEX-IR Basic Block

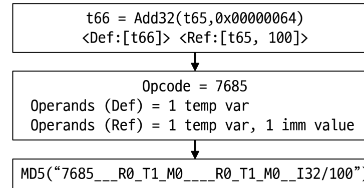

Fig. 5: Example of the Instruction Abstraction

Strand Generation Recall that a strand is a set of instructions in one basic block that is required to compute one variable. Therefore, one strand represents one independent data flow. The strand generator performs data-flow analysis by inspecting the define-reference dependencies among instructions within a basic block. Thanks to the SSA property of VEX-IR, we were able to deploy a simple algorithm to create strands, such as the one used in Esh [9]. We tweaked the characteristics of strands to improve accuracy and performance in two ways. First, one instruction can be included in only one strand to prevent duplication of instructions, which lowers overall accuracy by over-representing some instructions. Incidentally, the reduced number and size of strands helped the detector's

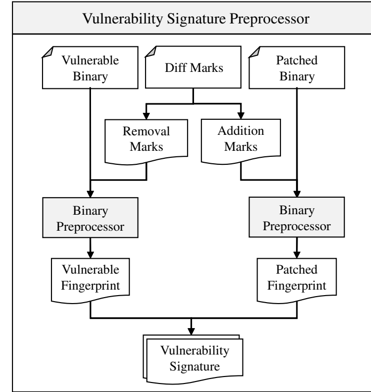

Fig. 6: Overview of the Vulnerability Signature Preprocessor

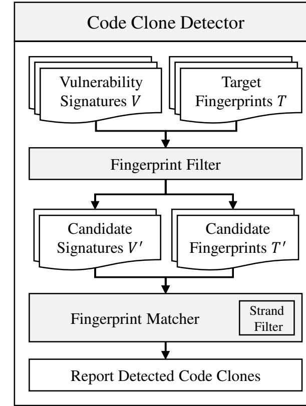

Fig. 7: Overview of the Code Clone Detector

performance. Second, a strand cannot span across multiple basic blocks. Splitting one long data flow improves performance by letting the code clone detector to be simplified.

After generating a strand, QuickBCC abstracts each instruction of the strand. This makes QuickBCC robust against changes to the compile environment, and enables fast comparison between two instructions. The change in the compile environment easily influences memory address offsets and registry allocation, so the instruction abstractor removes them. It leaves opcode and count of defined/referenced operands per operand type while keeping the values of immediates. Finally, each abstracted instruction is hashed with MD5. Fig. 5 illustrates the process of instruction abstraction. We chose MD5 as a hash function because MD5 had enough collision resilience for this task while being faster than other cryptographic hash functions.

3.3 Vulnerability Signature Generator

A vulnerability signature is an extended fingerprint, containing additional security patch information. The vulnerable signature preprocessor is built on the binary preprocessor. Fig. 6 represents the overview of the vulnerability signature preprocessor. It takes a pair of vulnerable and patched binaries along with a set of diff marks extracted from the binary diffing tool in Section 3.2. *Diff marks* represent the changes between vulnerable binary and patched binary. It consists of removed instructions from the vulnerable binary (a *removal mark*) and added instructions from the patched binary (an *addition mark*), the combination of which make up the security patch. Each set of marks is given to the binary preprocessor to track relevant strands from the vulnerable and patched fingerprints. This process helps QuickBCC locate the vulnerable and patched code in the target binaries.

3.4 Code Clone Detector

Fingerprint Matcher The matcher computes the similarity of two functions by making use of the n-gram similarity between two strands as a basis. We chose n-gram similarity because each n-gram captures correlations of a small range of adjacent instructions through the ordering within the n-gram.

When the detector operates in sub-function level granularity, the similarity between a target fingerprint and a signature fingerprint is defined based on the vulnerability score vs and patch score ps, as defined in Equation (6), (7), (8). Vulnerability score vs is defined as the similarity between a removal mark (rm) and a target fingerprint, and patched score ps is defined as the similarity between an addition mark (am) and a target fingerprint.

$$F_x = \{rm_x, am_x\} \qquad F_y = \{rm_y, am_y\} \tag{6}$$

$$vs = Sim(rm_x, rm_y) \qquad ps = Sim(am_x, am_y) \tag{7}$$

$$Sim(F_x, F_y) = vs - a \cdot ps \tag{8}$$

ps tends to have a higher false-positive rate than vs, so the coefficient a was given to mitigate the side effect for the worst-case scenario. We chose $a = 0.2$ in the evaluation.

Fingerprint & Strand Filtering When a user inputs a target binary, the matcher filters target fingerprints and vulnerability signatures, trimming out irrelevant ones. We define a *stable property* as a property of an entity within a binary that is minimally affected by a change of compile environments. We can find stable properties at each level of binary, such as a binary itself, a function, or a strand. Determining which properties are stable is crucial for the scalability of binary code clone detection. It allows us to reduce the search space we should test to find similar code fragments, improving the performance.

We leveraged stable properties as a filtering mechanism to find out if a target entity is irrelevant from a vulnerability signature. For example, the total count of external library function calls is not likely to change in different compile environments. Therefore, we depend on external call count to eliminate irrelevant target functions.

4 Evaluation

In Section 4, we show that QuickBCC can scan large volume of binaries with accuracy and high perfomance. Comparison with other approach is included later in this section.

4.1 Environmental Setup

Test Environment We performed all evaluations on an Ubuntu 20.04 LTS machine equipped with an Intel Core i5-9400 6-core 6-thread CPU. We built the framework with the .NET Core 3.1.10 and Python 3.8.5 runtime. When the tools we compared required a Windows environment, we evaluated it with a Windows 10 v20H2 virtual machine guest running on the same machine. Later, the OS of the machine is changed to Arch Linux when we evaluated accuracy of QuickBCC. But changing the OS of the machine did not affect accuracy.

Target Binaries We chose Debian Live amd64 GNOME ISO images from the past 3.5 years (versions 9.0.0 - 10.8.0) as the source of our binaries. We chose Debian because (1) Debian officially tracks vulnerability information of the packages in its security tracker [1], (2) which enabled us collecting signature binaries. We used it as a ground truth. We extracted binaries from the */bin*, */sbin*, and */usr* directories of the *filesystem.squashfs* from the live image, excluding symlink files. Table 1 shows the statistics of the target binary set.

debian-live-amd64-gnome (9.0.0 - 10.8.0)	
# Binary File	143,018
# Binary File Size (Total)	19.4GB
# Binary File Size (Avg Per File)	143.2KB
# Target Func (Total)	51,654,380
# Target Func (Avg Per File)	361.2
# Target Func After Filtering (Total)	3,228,042
# Target Func After Filtering (Avg Per File)	22.6

Table 1: Target binaries used in evaluation

OSS	CVE	Package	Vuln Version	Patched Version	Lang
OpenSSL	CVE-2018-0734	libssl1.1	1.1.0h-4	1.1.0j-1~deb9u1	C
	CVE-2019-1547	libssl1.1	1.1.0k-1~deb9u1	1.1.0l-1~deb9u1	
	CVE-2019-1563	libssl1.1	1.1.0k-1~deb9u1	1.1.0l-1~deb9u1	
WavPack	CVE-2018-19841	libwavpack	5.0.0-2+deb9u2	5.1.0.6	C
TagLib	CVE-2017-12678	libtag	1.11.1+dfsg.1-0.1	1.11.1+dfsg.1-0.3	C++
Binutils	CVE-2018-6543	binutils	2.30-2	2.30-3	C

Table 2: Vulnerable binaries used in evaluation

Vulnerability Signatures We collected real-world software for multimedia processing in popular use [5], especially pre-installed software (amd64) in the Debian distribution image. Table 2 shows the list of vulnerability signatures we collected.

Threshold We conducted an experiment to determine the threshold value. We checked the result by increasing the threshold value from 0.5 to 1 by 0.02. We used live images from Debian from the past two years, and tested with the 6 signatures mentioned in Table 2. Fig. 9 shows that if we use a threshold value of 0.84 or higher, we can see a corresponding precision value of 1.0 and high value of F1 score.

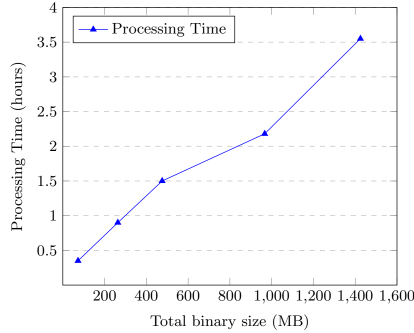

Fig. 8: Relation between total binary size and processing time

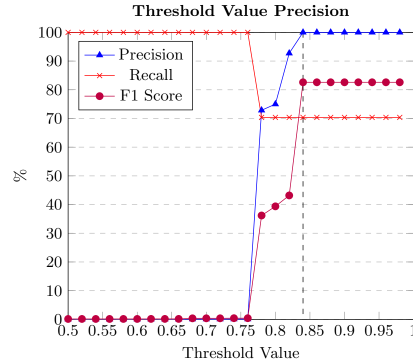

Fig. 9: Precision and Recall of each Threshold value

4.2 Vulnerable Code Clone Detection

We first matched the 6 vulnerability signatures we generated with Debian 9.0.0 live image to evaluate its performance. And later we matched the 6 vulnerability signatures whole target binaries with 52.6 million target functions from 139,446 target binaries to evaluate its accuracy.

Preprocessing + Matching (1st run)

Threads	1	6
Run Time	8h 53m 34s	1h 51m 33s
Preprocessing Time	8h 53m 19s	1h 51m 10s
Preprocessing Time (per file)	5692ms	-
Matching Time	15s	23s
Matching Time (per file)	3ms	-
Matching Time (per signature)	0.01ms	-

Cache Loading + Matching (2nd+ run)

Threads	1	6
Run Time	3m 24s	3m 9s
Cache Load Time	1m 34s	2m 5s
Cache Load Time (per file)	19ms	-
Matching Time	1m 50s	1m 4s
Matching Time (per file)	3ms	-
Matching Time (per signature)	0.01ms	-

Table 3: Performance of QuickBCC in single-threaded environment

Performance Table 3 shows the fast performance and high scalability of QuickBCC. The framework required 111 minutes to scan 5,439 files (from Debian 9.0.0 only) in the first run with 6 threads, which involved preprocessing and matching. The framework caches preprocessed target functions to accelerate later runs. When it loaded the cached fingerprints, it completed the overall scanning in 4 minutes. The ability to scan 2.1 million functions from an entire Linux desktop distribution in a few minutes effectively demonstrates the high scalability of QuickBCC.

The filtering system of QuickBCC also played a significant role in improving scalability. The framework filtered out 94.9% of the target functions before the actual matching was performed. As a result, it took only 59 seconds to match 6 vulnerability signatures with all 5,439 binary files and 11ms to scan each target binary with scalability(Fig. 8)

Target OS Version	Detected Signature	TP	FP	TN	FN
9.0.0	4	4	-	1	2
9.1.0	4	4	-	1	2
9.2.0	4	4	-	1	2
9.3.0	4	4	-	1	2
9.4.0	4	4	-	1	2
9.5.0	4	4	-	1	2
9.6.0	4	4	-	2	1
9.7.0	4	4	-	2	1
9.8.0	4	4	-	2	1
9.9.0	4	4	-	2	1
9.11.0	4	4	-	2	1
9.12.0	3	3	-	3	1
10.0.0	1	1	-	5	-
10.1.0	1	1	-	5	-
10.2.0	-	-	-	6	-
10.3.0	-	-	-	6	-
10.4.0	-	-	-	6	-
10.5.0	-	-	-	6	-
10.6.0	-	-	-	6	-
10.7.0	-	-	-	6	-
10.8.0	-	-	-	6	-

Table 4: Detected vulnerable code clones for each target OS

Accuracy Table 4 show our matching results. QuickBCC detected 49 vulnerable code clones from 6 target binaries. Detected clones consist of 49 true-positives, 0 false-positive, 71 true-negatives, and 18 false-negatives. We tested the framework with a threshold t of 0.84 and a coefficient a of 0.2. An analysis of each case is shown below.

OpenSSL Vulnerabilities We created the signatures of CVE-2019-1547 and CVE-2019-1563 from the OpenSSL 1.1.0k (vulnerable) and 1.1.0l (patched) binaries. Three branches of OpenSSL, including the 1.1.0 and 1.0.2 branches, had these vulnerabilities. As a result, QuickBCC detected two vulnerable code clones for CVE-2019-1563 vulnerability. Debian 9 includes OpenSSL 1.1.0 (*libcrypto.so.1.1*) and 1.0.2 (*libcrypto.so.1.0.2*) binaries, and both of them were found to be vulnerable. Therefore, three vulnerable code clones were detected for the two signatures CVE-2019-1547 and CVE-2019-1563, all of which were true-positives.

In contrast, the CVE-2018-0734 signature created false-negatives that the code clone was not found from the OpenSSL binaries. We had looked through the impacted OpenSSL source file and discovered the target binary did contain source code older than the binary we used to create the vulnerability signature. Since the target binary did not have an exact code pattern QuickBCC was searching for, we saw the framework worked as intended.

WavPack Vulnerability We found one true-positive vulnerability code clone of CVE-2018-19741 from a vulnerable libwavpack binary, while avoiding false-positives.

TagLib Vulnerability Since the patch of CVE-2017-12678 is changing casting and the changed binary is tiny, we got false negatives from the low similarity score.

Binutils Vulnerability No code clones were found from the CVE-2018-6543 signature. We had looked through the Binutils packages of the target and concluded

Signature	Target	Time (milliseconds)	
		Esh	**QuickBCC**
Heartbleed (GCC)	Heartbleed (GCC)	672	0.192
Heartbleed (Clang)	Heartbleed (Clang)	676	0.281
Shellshock (GCC)	Shellshock (GCC)	820	32.677
Shellshock (ICC)	Shellshock (ICC)	861	95.082
Average		**$757 \cdot 10^5$**	**32.08**

Table 5: Comparison with *Esh* about the time required to match one pair of strands

that it is a true-negative. The version of the binary used to create a vulnerability signature was different from the target binaries.

4.3 Comparison

We compared QuickBCC with *Esh* [9], which is the most recent and similarly designed system. Esh provides its matcher source code, its binary dataset, and four sample strands. We evaluated Esh by matching a pair of sample strands and measuring the comparison time. We could only use the provided strands with Esh, as we did not have access to its preprocessor. For QuickBCC, we preprocessed target object files from the Esh dataset and matched all strands of the target function with the signature function, calculating the matching time per strand.

Table 5 shows the results we found comparing Esh and QuickBCC. We found QuickBCC was about 22 times faster than Esh on average, with acceptable levels of accuracy. Esh took about 850ms to match a pair of strands from a Shellshock vulnerability, while QuickBCC took only about 60ms. It means that Esh takes more than a day to scan a large amounts of binaries such as Debian live image. The comparison shows the performance advantage of the QuickBCC framework. We achieved this by a fast method of comparing semantics with n-gram similarity and using extensive filtering.

5 Discussion & Future Work

5.1 Robustness to Multiple Compile Environments

We proved QuickBCC worked well when we controlled the compile environments to make signatures in each evaluation, but some elements of the compile environment were hard to mitigate. For example, QuickBCC had an issue matching binaries from two different platforms. We propose to solve this problem by preparing a set of signatures per vulnerability, where each signature stands for one compile environment. It is a reasonable workaround based on what we have seen. First, running the matcher multiple times is not a significant burden, thanks to its filtering and high performance. Second, merging several similar signatures into one signature would significantly reduce the number of required signatures. Finally, the practical number of required vulnerability signatures is not very high, since most platforms use a limited number of compile environments.

For example, We can create minimal signatures per vulnerability that they can span signatures from almost all compiler options. In the case of CVE-2019-1563, a signature compiled with O3 can detect vulnerabilities in unpatched binary that compiled with O1, O2, O3, Og, and Ofast. The signature compiled with O0, though, could only detect unpatched binaries that compiled with O0. In this case, only 1/3 of the signature (O0, O3) could cover all optimization levels.

5.2 Better Vulnerability Signature Generation

Employing a better binary diffing method would improve QuickBCC's accuracy. Conventional binary diffing algorithms used in QuickBCC mark instructions moved by a patch or compile environment difference as deleted or added instructions. It can cause false-negatives on vulnerable code clone detection, so a user has to review and fix the diff results. It would be useful to create a binary diffing method specific to QuickBCC's vulnerability signature generator.

Next, automatically creating vulnerability signatures would be another milestone of this research. We experimented vulnerability signature auto-generation by exploiting binary debug symbols, but that method had limitations. Compiler optimization distorted the mapping between the source patch line and the binary instruction offset, reducing the accuracy of generated signatures. We expect that combining our effort with the conventional binary diffing method would be a breakthrough.

6 Conclusion

We presented QuickBCC, a scalable binary code clone detection framework for practical vulnerability scanning. We managed to extract and statically compare semantics, to achieve high performance and accuracy. We also leveraged the notion of strands and introduced a similarity metric for ordered sets to the world of binary code clones. We invented the removal mark and the addition mark to enable effective vulnerable code clone detection for binaries. QuickBCC demonstrated its fast performance with extensive filtering and code optimization, while it proved to be accurate enough to detect a vulnerable code clone from a similar yet different set of binaries.

Acknowledgement

We thank Seongbeom Park for his contribution on the signature generation. This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01697, Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security), the National Research Foundation (NRF), Korea, under project BK21 FOUR, and the Research Foundation City University of New York.

References

1. Debian Security Tracker. https://security-tracker.debian.org/tracker (1997-2021), accessed on 2021-04-15
2. VEX-IR. https://sourceware.org/git/?p=valgrind.git;f=VEX/pub/libvex_ir.h;a=blob_plain (2004-2021), accessed on 2021-04-15
3. IDA: About. https://www.hex-rays.com/products/ida/ (2005-2021), accessed on 2021-04-15
4. Diaphora, the most advanced Free and Open Source program diffing tool. https://github.com/joxeankoret/diaphora (2015-2021), accessed on 2021-04-15
5. (2021), https://popcon.debian.org/
6. radare. https://rada.re/n/radare2.html (2021), accessed 2021-04-15
7. Alrabaee, S.: Efficient, Scalable, and Accurate Program Fingerprinting in Binary Code. Ph.D. thesis, Concordia University (2018)
8. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and Evaluation of Clone Detection Tools. IEEE Transactions on Software Engineering **33**(9), 577–591 (Sep 2007). https://doi.org/10.1109/TSE.2007.70725
9. David, Y., Partush, N., Yahav, E.: Statistical Similarity of Binaries. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 266–280. PLDI '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908126
10. David, Y., Partush, N., Yahav, E.: FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In: ASPLOS'18. pp. 392–404 (May 2018). https://doi.org/10.1109/SP.2017.62
11. Ding, S.H., Fung, B.C., Charland, P.: Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 472–489. IEEE (2019)
12. Duan, Y., Li, X., Wang, J., Yin, H.: Deepbindiff: Learning program-wide code representations for binary diffing. In: Proceedings of the Network and Distributed System Security Symposium (2020)
13. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In: Proceedings of the 2016 Network and Distributed System Security (NDSS) Symposium (2016)
14. Feng, Q., Wang, M., Zhang, M., Zhou, R., Henderson, A., Yin, H.: Extracting conditional formulas for cross-platform bug search. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. pp. 346–359 (2017)
15. Gao, D., Reiter, M.K., Song, D.: Binhunt: Automatically finding semantic differences in binary programs. In: International Conference on Information and Communications Security. pp. 238–255. Springer (2008)
16. Huang, H., Youssef, A.M., Debbabi, M.: BinSequence: fast, accurate and scalable binary code reuse detection. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. pp. 155–166. ACM (2017)
17. Jang, J., Agrawal, A., Brumley, D.: ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In: 2012 IEEE Symposium on Security and Privacy. pp. 48–62 (May 2012). https://doi.org/10.1109/SP.2012.13
18. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7), 654–670 (2002)

19. Kim, S., Woo, S., Lee, H., Oh, H.: VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 595–614 (May 2017). https://doi.org/10.1109/SP.2017.62
20. Kondrak, G.: N-gram Similarity and Distance. In: Proceedings of the 12th International Conference on String Processing and Information Retrieval. pp. 115–126. SPIRE'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11575832_13, http://dx.doi.org/10.1007/11575832_13
21. Massarelli, L., Di Luna, G.A., Petroni, F., Baldoni, R., Querzoni, L.: Safe: Self-attentive function embeddings for binary similarity. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 309–329. Springer (2019)
22. Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. SIGPLAN Not. **42**(6), 89–100 (Jun 2007). https://doi.org/10.1145/1273442.1250746
23. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T.: Cross-Architecture Bug Search in Binary Executables. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy. pp. 709–724. SP '15, IEEE Computer Society, Washington, DC, USA (2015). https://doi.org/10.1109/SP.2015.49, https://doi.org/10.1109/SP.2015.49
24. Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., Debbabi, M.: Bincomp: A stratified approach to compiler provenance attribution. Digital Investigation **14**, S146–S155 (2015)
25. Shirani, P., Collard, L., Agba, B.L., Lebel, B., Debbabi, M., Wang, L., Hanna, A.: Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 114–138. Springer (2018)
26. Xiao, Y., Chen, B., Yu, C., Xu, Z., Yuan, Z., Li, F., Liu, B., Liu, Y., Huo, W., Zou, W., et al.: {MVP}: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In: 29th USENIX Security Symposium (USENIX Security '20). pp. 1165–1182 (2020)
27. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 363–376 (2017)
28. Xu, Z., Chen, B., Chandramohan, M., Liu, Y., Song, F.: Spain: security patch analysis for binaries towards understanding the pain and pills. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 462–472. IEEE (2017)
29. Zhang, H., Qian, Z.: Precise and accurate patch presence test for binaries. In: 27th USENIX Security Symposium (USENIX Security '18). pp. 887–902 (2018)
30. Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., Zhang, Z.: Neural machine translation inspired binary code similarity comparison beyond function pairs. arXiv preprint arXiv:1808.04706 (2018)