Harm-DoS: Hash Algorithm Replacement for Mitigating Denial-of-Service Vulnerabilities in Binary Executables

Nicolaas Weideman
nweidema@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Haoda Wang
haodawan@usc.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Tyler Kann
kannt@uw.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Spencer Zahabizadeh
szahabiz@usc.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Wei-Cheng Wu
wwu@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Rajat Tandon
rajattan@usc.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Jelena Mirkovic
mirkovic@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Christophe Hauser
hauser@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

ABSTRACT

Programs and services relying on weak hash algorithms as part of their hash table implementations are vulnerable to hash-collision denial-of-service attacks. In the context of such an attack, the attacker sends a series of program inputs leading to hash collisions. In the best case, this slows down the execution and processing for all requests, and in the worst case it renders the program or service unavailable. We propose a new binary program analysis approach to automatically detect weak hash functions and patch vulnerable binary programs, by replacing the weak hash function with a secure alternative. To verify that our mitigation strategy does not break program functionality, we design and leverage multiple stages of static analysis and symbolic execution, which demonstrate that the patched code performs equivalently to the original code, but does not suffer from the same vulnerability. We analyze 105, 831 real-world programs and confirm the use of 796 weak hash functions in the same number of programs. We successfully replace 759 of these in a non-disruptive manner. The entire process is automated. Among the real-world programs analyzed, we discovered, disclosed and mitigated a zero-day hash-collision vulnerability in Reddit.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; Denial-of-service attacks.

RAID 2022, October 26–28, 2022, Limassol, Cyprus
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9704-9/22/10.
https://doi.org/10.1145/3545948.3545967

KEYWORDS

Binary program analysis, automatic vulnerability detection, automatic vulnerability mitigation

ACM Reference Format:
Nicolaas Weideman, Haoda Wang, Tyler Kann, Spencer Zahabizadeh, Wei-Cheng Wu, Rajat Tandon, Jelena Mirkovic, and Christophe Hauser. 2022. Harm-DoS: Hash Algorithm Replacement for Mitigating Denial-of-Service Vulnerabilities in Binary Executables. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3545948.3545967

1 INTRODUCTION

Denial-of-service (DoS) attacks can create significant losses to businesses, by slowing down processing of client requests or by making a service unavailable. There are many types of DoS attacks. In this paper, we focus on a *hash-collision DoS attack*, a type of algorithmic complexity attack [12], which exploits a vulnerability in weak hash table implementations in order to disrupt the availability of a target program.

Hash tables are data structures, ubiquitous for their fast, constant-time insertion and lookup operations. Because speed is at stake, hash table implementations typically use simple hash algorithms, which unfortunately also have low collision resistance. We denote these as *weak hash algorithms.* Attackers can easily generate inputs that will create collisions in hash tables that use weak hash algorithms. During a hash-collision DoS attack, the attacker crafts a large number of malicious inputs that are all inserted at the same table index, which drastically increases both the lookup and the insertion time. On each insertion and retrieval at the affected index, the hash table now has to iterate over a large list of colliding entries. This makes the operation time effectively linear in the number of entries. Likewise, inserting a number of these colliding entries requires polynomial time, allowing an attacker to greatly increase the computational load.

Hash-collision vulnerabilities have been discovered in the hash table implementation of the programming languages PHP [13], Python [14], and Java [15], and have affected all programs written in the vulnerable versions of these languages. This is especially critical when a remote attacker has control over hash table entries, as was the case with a PHP web server [13]. In addition, we have identified a remotely-exploitable zero-day hash-collision vulnerability, discussed in Section 10.3, using *Harm-DoS*. It is evident that the real world impact of such vulnerabilities is serious and has therefore attracted the attention of the security community in recent years.

Because algorithmic complexity vulnerabilities are a serious threat to security, there has been recent work in detecting these vulnerabilities automatically via static analysis, as presented by Kirrage [30] and Chang [11], as well as via fuzzing, presented by Petsios [39] and Blair [9]. Previous approaches based on static analysis do not focus on vulnerabilities caused by hash collisions and thus are likely to be less accurate in detecting them than *Harm-DoS*. Fuzzing, on the other hand, requires an extensive run time to find malicious inputs, which could be very sparse for hash functions. Thus fuzzing is unsuitable to use for detection of hash-collision vulnerabilities at scale. There has also been work on mitigating algorithmic complexity vulnerabilities by closing network connections that exploit a vulnerability as presented by Meng [36]. This approach is useful, but ideally, we would like to patch the vulnerable code to fully remove the vulnerability.

With the prevalence of proprietary, third-party software libraries, it is essential to conduct vulnerability analysis on binary code. Unfortunately, the problem of detecting and patching hash-collision vulnerabilities in executable programs, without relying on source code, has received little to no attention. We propose *Harm-DoS*, a novel approach to fill this gap by detecting and replacing weak hash functions automatically, at the binary level. In spite of the inherent complexity of working with binary code, *Harm-DoS* surgically analyzes the program to diagnose hash-collision vulnerabilities and perform a *hash transplant* – replacing the weak hash algorithm with a secure alternative. Similar to a medical organ transplant, the entire process must be conducted with utmost precision. We introduce *hash-collision vulnerability diagnosis*, a novel static analysis inspired by past research in detecting cryptographic hash functions by Lestringant and Gröbert, respectively [26, 33], and adapted to detect weak hash functions at scale. After diagnosis, *Harm-DoS* conducts a thorough *pre-patch examination*, a novel use of symbolic execution, to ensure the patch can be performed safely, without introducing critical errors (like accessing memory out of bounds). Next, *Harm-DoS* performs the hash transplant by leveraging static binary rewriting to replace the weak hash function with an appropriate secure alternative, crafted with the insights gained from the pre-patch examination. Finally, *Harm-DoS* conducts a *post-patch examination*, a second phase of symbolic execution to confirm that the replacement was successful and no errors were introduced. Since the weak hash function is removed from the patched program, the program is now resilient against hash-collision DoS attacks. *Harm-DoS* does not rely on source code or debug symbols, and simply requires the binary image of an executable program as input. The entire approach is automated.

To the best of our knowledge, our approach is the first to propose an automated solution to patch vulnerable hash algorithms at the binary level. We make the following contributions:

- We introduce the concepts of *hash-collision vulnerability diagnosis* and *hash transplant*, new approaches to automatically detect and replace weak hash algorithms in binary code.
- In the new concepts of *pre-patch* and *post-patch examination*, we leverage static analysis and symbolic execution in a novel way, along with insights tailored for non-disruptive patching, to preserve the original program semantics through verification steps.
- We implement a prototype of the proposed analysis, which is available as open source at https://github.com/usc-isi-bass/hashdos_vulnerability_detection.
- We evaluate our approach on 105, 831 binaries from the All-Star data set [45]. *Harm-DoS* confirms the use of 796 weak hash functions, in the same number of programs, and successfully replaces 759 (95%) of these in a non-disruptive manner.

2 SCOPE

Our work focuses on the detection and non-disruptive patching of weak hash algorithms, by replacing them with a secure alternative.

2.1 Fast Hash Algorithms

Many programs implement hash tables to store and process user input and internal data. Programs use hash functions to calculate an index for the hash table. While there are well-known cryptographic hash algorithms, which are collision resistant (e.g. MD5 and SHA256), these typically impose a significant performance penalty which make them unsuitable to be used in hash table implementations. Instead, programs use simpler hash algorithms, to achieve high performance, which we refer to as *fast hash algorithms*.

Due to the intricacies of developing a good hash algorithm, developers often reuse existing source-code implementations of well-known fast hash algorithms, with good average-case performance. In order to be reusable, the algorithm is often implemented in a single function, a *hash function*, that is context-agnostic, i.e. does not rely on any program specific assumptions. This means, hash functions are usually implemented without side effects, i.e. they do not modify the program state beyond the context of the function. This is beneficial to *Harm-DoS* because, it means it is possible to replace one such algorithm with another, without introducing unwanted side effects into the program.

Moreover, to be context-agnostic, many fast hash functions share common features. *Harm-DoS* relies on these features to identify fast hash functions in binary code. To receive input in a context-agnostic way, a hash function often receives a variable-length input buffer as a simple byte array, passed as function input parameter. We have observed that, in practice, many hash functions implement one of two signatures. We refer to these signatures as the *buffer-length* signature (the function receives a pointer to the byte array and the buffer's length),

```
unsigned int hash(const char* str, unsigned int length);
```

and the *buffer-only* signature (the function receives a pointer to the null-terminated byte array), `unsigned int hash(const char* str);`.

The hash function iterates over the bytes of the input array to compute a small integer (fitting within the bit-width of the architecture) – the *hash value*. At the start of a hash function, the hash value is initialized, often to a constant unique to the algorithm. On each iteration, the hash value is updated according to the definition of the algorithm. The final hash value is returned from the function. We show an example of such a hash function in Listing 4, in Appendix C.

During compilation, the context-agnostic nature of hash functions is often disrupted by an optimization feature called *function inlining*. Here the body of the hash function is placed directly in the body of the context-specific caller function. Patching such functions is out of scope for *Harm-DoS*. We discuss this decision in Sections 3 and 11.

2.2 Hash-Collision Vulnerabilities

The high performance guarantees of hash tables rely on a hash function that distribute the entries evenly over the table. An algorithm that is too simple can make it very easy for an attacker familiar with it to calculate colliding inputs. We refer to these as *weak* hash algorithms. When the attacker serves such inputs to an online program, they will degrade performance of the hash table from constant to linear time, and lead to denial of service.

A program can also use an internal secret within a fast hash algorithm to achieve collision resistance from an attacker, discussed by Aumasson and Alakuijala, respectively [1, 3], while maintaining computational efficiency. We refer to these as *secure* hash algorithms, which we use to replace the weak hash algorithm, when mitigating hash-collision vulnerabilities.

2.3 Attacker Model

We assume that a remote attacker knows which hash algorithm is used by the target program (i.e., they can obtain an official version of the program). We also assume the remote attacker can observe inputs and outputs of the target program (e.g., by interacting through a network socket) but cannot observe intermediate computations or the internal state of the program (which would require local access with debugging capabilities). This assumption is reasonable since programs generally use hashes internally, e.g., as an index into a hash table, and do not disclose these values to the user. Finally, we consider an attacker obtaining information about either the hash value, or internal program state via side-channel attack to be out of scope.

3 CHALLENGES AND REQUIREMENTS

The first step in mitigating hash-collision vulnerabilities automatically is to detect them. This is challenging, since a program could use any arbitrary hash algorithm. As with any program analysis approach, there are strict theoretical limits to what can be determined regarding the behavior of the target binary. Therefore, identifying any possible weak hash algorithm is infeasible. Instead, *Harm-DoS* focuses on detecting implementations of several known-weak hash algorithms, which we call *weak hash functions*.

We aim to detect weak hash functions in stripped binary executables, meaning we cannot rely on human-friendly artifacts of source code (e.g., function and variables' names and comments) in order to derive information regarding the purpose of different pieces of code. To overcome this challenge, we instead rely on the computations and control flow inherent to a list of known-weak hash algorithms to detect the vulnerability. This is explained in Section 5.

If a hash function is inlined, the boundaries between it and its caller function are blurred. Identifying the function boundaries of inlined functions is a research problem orthogonal to the focus of *Harm-DoS*, as done by Bao [4]. *Harm-DoS* detects inlined hash functions, but we leave patching them for future work (Section 11).

After detecting the vulnerability, the next challenge is to mitigate it automatically. Patching hash functions at the source code level usually involves replacing the weak hash function (the *original hash function*) with a secure alternative (the *replacement hash function*). Indeed, this was the approach taken in mitigating the hash-collision vulnerability in Python [14, 27] as well as Perl [38]. We reproduce this process in binary code, but encounter several challenges because we must modify the binary code, while preserving its correctness.

In order to address the outlined challenges effectively, *Harm-DoS* must fulfill the following vulnerability detection requirements (DR) as well as mitigation requirements (MR).

DR1: It is important to modify only the binary code instructions that form part of an implementation of a weak hash function. It is therefore critical that *Harm-DoS* identifies weak hash functions correctly. We require that *Harm-DoS* has zero false positives among the successfully patched functions. In Sections 5 and 6.2.2 we discuss our strategy for detecting and confirming the detection of weak hash functions.

DR2: Inlined hash functions in a given executable should either all be patched or none should be patched. Replacing only some of these would result in a hash table using different hash algorithms in different scenarios and, in turn, would break the functionality. We address the unique challenges presented by inlined hash functions partially, by identifying the presence of inlined hash algorithms and leaving these unpatched. Therefore, we require *Harm-DoS* to be able to identify inlined functions, as discussed in Section 6.1.

If *Harm-DoS* replaces a weak hash function, we need to preserve correctness with regard to how inputs are processed, the range of the outputs produced, and the functionality of the rest of the program code. To achieve this *Harm-DoS* must fulfill the following mitigation requirements.

MR1: *Harm-DoS* must only replace the binary code responsible for implementing the weak hash algorithm. Replacing any other instructions will introduce defects into the program. We discuss our strategy for replacing the hash function in Section 7.2

MR2: To ensure *Harm-DoS* does not affect the program behavior in unintended ways, we require the replacement hash function to restore the program to its prior state after completion, apart from the hash value. We make an exception here for the stack memory, local to the replacement hash function, since this memory is discarded once the function returns. We discuss this in Section 7.1.

MR3: *Harm-DoS* must be able to detect side effects in the original hash function. We identify two types of such side effects, namely when the hash function writes to global memory, or when it writes to a memory address passed via input parameter. Replacing the hash function while omitting these side effects will affect the program behavior in an unknowable way. Therefore, *Harm-DoS* must detect these side effects and not replace the weak hash function. The method for achieving this is discussed in Section 6.2.4.

MR4: In order to avoid illegal memory accesses, the replacement hash function must not access any memory that is not accessed by the original. We make an exception for the stack memory of the replacement hash function. We discuss this in Section 8.1.

MR5: The replacement hash function must return hash values that are no greater than the maximum of the original hash function. As the hash value is often used to calculate an index in a hash table, yielding larger hash values than expected may lead to memory access errors. Our approach to address this problem is discussed in Sections 6.2.2 and 8.2.

MR6: The replacement hash function must consume its input in the same way as the original hash function. This is discussed in Section 6.2.1.

MR7: The replacement hash function must match the original in terms of case sensitivity. A case-insensitive hash function yields equal hash values for input buffers that differ only in case, e.g. buffers `abc` and `AbC`. Using a case-sensitive hash function to replace a case-insensitive hash function will cause incorrect lookups in the hash table. Conversely, using a case-insensitive hash function to replace a case-sensitive hash function makes exploitation trivial, even for secure hash functions. We discuss the solution to this requirement in Section 6.2.3.

MR8: The replacement hash function introduced by *Harm-DoS* must be *secure*, i.e., collision resistant with respect to our definition in Section 2.2 and our attacker model. We discuss such replacement hash functions in Section 7.1.1.

4 APPROACH OVERVIEW

Harm-DoS is divided into five phases, shown in Figure 1, namely *Analysis Preparation*, *Vulnerability Diagnosis*, *Pre-patch Examination*, *Hash Transplant*, and *Post-patch Examination*. *Harm-DoS* receives as input a binary executable file, the *target binary* as well as a set of *detection models*, where each detection model contains the features necessary to detect a specific weak hash algorithm. In preparation for analysis, *Harm-DoS* disassembles the target binary, recovers control flow and identifies the function boundaries therein.

In the Vulnerability Diagnosis phase, *Harm-DoS* analyzes functions in the target binary. We refer to the function under analysis as the *target function*. In the target function, *Harm-DoS* detects the presence of a weak hash algorithm towards fulfilling **DR1** and **DR2**. This phase leverages static analysis of both the instructions and control-flow of the binary code of the target function. First, the target function is matched against a hash function template, designed to detect the presence of patchable hash functions, with the features described in Section 2.1. If a *template-match* is found, we compare the function to each detection model created for a known-weak hash algorithm. This light-weight static analysis allows us to pinpoint *candidate hash functions* quickly, in linear execution time in the size of the target binary. *Harm-DoS* will analyze these candidate hash function with a more accurate, but more expensive analysis.

With the set of candidate hash functions, *Harm-DoS* proceeds to determine those functions that can be patched, while preserving correctness, in the Pre-patch Examination phase. *Harm-DoS* employs static analysis to determine if the hash algorithm is *isolated* in a function, away from other functionality, for **DR2** and **MR1**. *Harm-DoS* also employs symbolic execution to build a profile of the behavior of each candidate hash function, with regards to signature (**MR6**), input-output relationships (**DR1**, **MR5**), case sensitivity (**MR7**), and memory accesses (**MR3**). *Harm-DoS* preserves this profile during patching. The input-output relationships are of particular importance, as these enable *Harm-DoS* to confirm that the candidate hash function has indeed been identified correctly. We refer to such candidate hash functions as *confirmed hash functions*.

For the confirmed hash functions for which *Harm-DoS* can preserve correctness, *Harm-DoS* performs the Hash Transplant to mitigate the vulnerability, by replacing the hash algorithm with a secure alternative, while fulfilling **MR2**, **MR1** and **MR8**, by modifying the binary executable. The patched executable is passed to the final phase of *Harm-DoS*, Post-patch Examination. In this phase the memory accesses and output values of the replacement hash function are monitored to confirm that correctness has indeed been preserved, fulfilling **MR4** and **MR5**. If this verification passes, we consider the patch a success, otherwise the patch is discarded and an error is reported.

5 VULNERABILITY DIAGNOSIS

In this section, we provide more detail on how we discover candidate hash functions in target binaries, for **DR1** and **DR2**. While there is significant past research in code similarity and detecting cryptographic algorithms, such as done by Lestringant [33], Gröbert [26], Farhadi [18] and Bruschi [10, 47], the focus of these often lie in detecting a specific implementation of an algorithm. *Harm-DoS*, on the other hand, strikes a balance between being implementation agnostic and identifying those weak hash functions that can be patched. This is achieved with a novel approach that is simple, yet effective. This approach uses static analysis to determine how a target function interacts with the program memory, while making minimal assumptions with regards to how a hash algorithm is implemented. Consequently, *Harm-DoS* detects candidate hash functions optimistically. This is beneficial for **DR2**, for which it is important to detect all implementations of a single hash algorithm in a target binary. Even though this approach leads to false positive detections, these will be pruned in Pre-patch Examination, preventing a faulty patch.

Harm-DoS performs Hash Function template-matching (Section 5.1) to determine if the target function matches a template, designed from the insights in Section 2.1. For any matching target function, *Harm-DoS* determines if it implements a known-weak hash algorithm using Constant-Mnemonic Pair Discovery, Section 5.2. For each known-weak hash algorithm, *Harm-DoS* uses a supplied *detection model* that captures the constants used in the calculations defined in the algorithm. We create a detection model for each of the following list of popular known-weak hash algorithms: BKDR [29], DEK [31], DJB [8], ELF [25], FNV [21], JS [40],

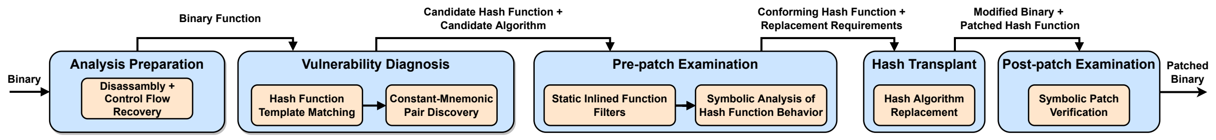

Binary
Analysis Preparation
Disassambly + Control Flow Recovery
Binary Function
Vulnerability Diagnosis
Hash Function Template Matching
Constant-Mnemonic Pair Discovery
Candidate Hash Function + Candidate Algorithm
Pre-patch Examination
Static Inlined Function Filters
Symbolic Analysis of Hash Function Behavior
Conforming Hash Function + Replacement Requirements
Hash Transplant
Hash Algorithm Replacement
Modified Binary + Patched Hash Function
Post-patch Examination
Symbolic Patch Verification
Patched Binary

Figure 1: A flowchart of the approach overview.

RS [43], and SDBM [42]. Note that this list can be easily extended to other weak hash algorithms. The outcome of this phase is a set of *candidate hash functions* in the target binary, each with a *candidate algorithm* – a label indicating which known-weak hash algorithm we suspect is implemented therein (e.g., BKDR).

5.1 Hash Function Template-matching

The first step of identifying candidate hash functions, is to determine if a target function f contains the features of a weak hash function. In Section 2.1, we mention that weak hash algorithms typically iterate over a variable-length input buffer. Iterating over such a buffer necessarily manifests as a loop in control flow. Therefore, *Harm-DoS* requires a target function to have at least one nontrivial strongly connected component (SCC)[1] in its control-flow graph (CFG). Let $\text{CFG}(f) = G_f$ denote the CFG of f and let $\text{SCC}(G_f)$ denote the set of nontrivial SCCs $\{C_{f,1}, C_{f,2}, \ldots, C_{f,M}\}$ of G_f. *Harm-DoS*, therefore, requires $|\text{SCC}(G_f)| \geq 1$.

Next, *Harm-DoS* analyzes the instructions of each SCC to identify possible side effects of the target function. As mentioned in Section 2.1, weak hash functions are often implemented without side effects in order to be context-agnostic and a function with side effects cannot be replaced while fulfilling **MR3**.

Let $V(G)$ and $E(G)$ denote the set of nodes and edges in graph G, respectively. In G_f the nodes $b_1, b_2, \ldots, b_N$ are basic blocks and the edges represent the control-flow between them. Each basic block b_i is a sequence of instructions where $b_{i,j}$ denotes the j-th instruction. *Harm-DoS* identifies side effects by identifying instructions that perform a memory-write operation to memory outside of the stack memory of the target function. Memory-write operations receive the destination memory address as an expression consisting of registers, constants, or both. *Harm-DoS* analyzes this expression to determine whether a specific memory-write operation indicates a side effect. For an instruction $b_{i,j}$, $\text{WR}_\text{r}(b_{i,j})$ and $\text{WR}_\text{c}(b_{i,j})$ denote the set of registers and constants, respectively, used in the expression of the address in a memory write operation. If $b_{i,j}$ does not write to memory, $\text{WR}_\text{r}(b_{i,j}) = \text{WR}_\text{c}(b_{i,j}) = \emptyset$. Write operations to memory inside the stack memory are usually identifiable as a write to an offset from a register holding the stack pointer or the stack base pointer. Let $\mathcal{R}_\text{stack}$ denote the registers used in stack operations. In AMD64, $\mathcal{R}_\text{stack} = \{$`rsp`, `rbp`$\}$. Therefore, *Harm-DoS* explicitly only allows such memory-write operations in an SCC. Conversely, other types of memory-write operations, that use nonstack registers, or use only constants in the destination address expression are forbidden. For example, the instruction `mov [rsp-8],rax` is allowed, while instructions `xor [rsi+4],rbx`; `mov [rsp+rcx],rsi` and `mov [10000],rdi` are forbidden.

[1]A nontrivial SCC is a subgraph of mutually reachable nodes, with at least one node and edge (we allow self loops).

We say an SCC $C_{f,i}$ is a *template-match* if every memory write operation in the SCC writes only to memory on the stack. That is,

$$\forall b_j \in V(C_{f,i}),\ \forall b_{j,k} \in b_j,$$
$$(|\text{WR}_\text{c}(b_{j,k})| > 0 \implies |\text{WR}_\text{r}(b_{j,k})| > 0) \land \text{WR}_\text{r}(b_{j,k}) \subseteq \mathcal{R}_\text{stack}$$

We say a function f is *template-matching* if it contains an SCC that is a template-match.

5.2 Constant-Mnemonic Pair Discovery

Given a template-matching function, the next step is to determine if it is a known-weak hash function. To this end, *Harm-DoS* leverages a code identification technique relying on unique constants, proposed by Lestringant [33]. We include the unique constants found in weak hash algorithms paired with their assembly-language operators (mnemonics) in the detection model of each known-weak hash algorithm. For example, in order for a target function to be considered an SDBM hash function, it must either contain the constant 65599 used with the signed multiplication operator (`imul`), or both the constants 6 and 16, each used with the logical left shift operator (`shl`). This is because, in some implementations of this algorithm the calculation performed by the algorithm is implemented as `h * 65599`, while in others as `(h << 6) + (h << 16) - h`[2].

Formally, in the detection model of each known-weak hash algorithm, we include a set of *constant-mnemonic pair fingerprints.* For a hash algorithm A, let $\text{CMF}(A) = \{\mathcal{S}_{\text{cm}_1}, \mathcal{S}_{\text{cm}_2}, \ldots, \mathcal{S}_{\text{cm}_n}\}$ be a set of constant-mnemonic pair fingerprints. Each constant-mnemonic pair fingerprint $\mathcal{S}_{\text{cm}_i}$ is a set of tuples $\{(c_{i,1}, \mathcal{S}_{\text{m}_{i,1}}), (c_{i,2}, \mathcal{S}_{\text{m}_{i,2}}), \ldots\}$, where each $c_{i,j}$ is a constant and each $\mathcal{S}_{\text{m}_{i,j}}$ is a set of mnemonics $\{m_1, m_2, \ldots\}$. These act like fingerprints in the sense that they are unique to one known-weak hash algorithm. Due to the simplistic nature of weak hash algorithms, some have very few algorithm-specific constant-mnemonic pairs to use for identification. We opt for an optimistic approach, leading to false positives which are subsequently filtered out in the Pre-patch Examination phase.

Harm-DoS searches for the presence of these constant-mnemonic pair fingerprints in the instructions of the target function f. For a function f, let $\text{CM}(f)$ denote the set of tuples $\{(c_1, m_1), (c_2, m_2), \ldots\}$ where each tuple (c_i, m_i) represents a mnemonic m_i and constant operand c_i used in the instructions of f. We say f has a *constant-mnemonic pair match* with respect to weak hash algorithm A, if all the constants of a constant-mnemonic pair fingerprint of A appear within f, each used with one of its paired mnemonics. That is,

$$\exists\, \mathcal{S}_{\text{cm}_{i'}} \in \text{CMF}(A) \mid \forall (c_{i',j}, \mathcal{S}_{m_{i',j}}) \in \mathcal{S}_{\text{cm}_{i'}},\ \exists m_k \in \mathcal{S}_{m_{i',j}} \mid (c_{i',j}, m_k) \in \text{CM}(f)$$

[2]Note these implementations are mathematically equivalent.

Table 1: The constant-mnemonics pair fingerprints of each known-weak hash algorithm.

Hash	Constant-Mnemonic Pair Fingerprints
BKDR	{(131, {imul, mov})}, {(1313, {imul, mov})}
DEK	{(5, {rol})}, {(5, {shl}), (27, {shr})}
DJB	{(5381, {imul, mov})}
ELF	{(4, {shl}), (24, {shr, sar}), (4026531840, {and, mov})}
FNV	{(16777619, {imul, mov}), (2166136261, {mov})}
JS	{(2, {shr}), (5, {shl}), (1315423911, {mov})}
RS	{(63689, {mov}), (378551, {mov, imul})}
SDBM	{(65599, {imul})}, {(6, {shl}), (16, {shl})}}
	{(6, {shl}), (10, {shl})}

We generate the constant-mnemonic pair fingerprints, shown in Table 1, by compiling source code implementations of the known-weak hash algorithms with compilers GCC-7.5.0 and Clang-6.0.0. We use optimization levels `O0`, `O1`, `O2`, `O3`, `Os`, `Ofast` and `O0`, `O1`, `O2`, `O3`, `Os`, `Ofast`, `Oz` `Og`, respectively.

If the target function f is template-matching and has a constant-mnemonic pair match with respect to algorithm A, we say f is a *candidate hash function* with *candidate algorithm A*, denoted f_{c_A}. We show a full example of discovering a candidate hash function in Appendix A. Candidate hash functions must be analyzed to determine patchability.

6 PRE-PATCH EXAMINATION

In order to patch the candidate hash function, it is important to understand how it interacts with the rest of the program, in terms of control flow and data flow. This phase starts by identifying and filtering inlined hash algorithms in accordance to **DR2**, by using a heuristic static analysis of two steps Duplicate Candidate Algorithm Detection (Section 6.1.1) and Template-match Size Difference (Section 6.1.2). Candidate hash functions that remain after this step, are referred to as *isolated hash functions*. Focusing on isolated hash functions, allows us to use symbolic execution to build a behavior profile of the hash function.

Harm-DoS continues by applying four analysis steps, Symbolic Signature Detection (Section 6.2.1), Symbolic Input-Output Matching (Section 6.2.2), Symbolic Case Sensitivity Checking (Section 6.2.3) and Symbolic Memory Access Analysis (Section 6.2.4). These steps leverage symbolic execution on each of the isolated hash functions in the set to build a profile of their behavior.

6.1 Filtering Inlined Hash Functions

The hash function discovery analysis described in Section 5 makes no distinction between whether a hash algorithm has been inlined or not. In this section, we describe the static analysis we use to remove inlined hash algorithms from the set of candidate hash functions, fulfilling **DR2**.

6.1.1 Duplicate Candidate Algorithm Detection. If a hash algorithm A is inlined in multiple caller functions, the hash function discovery analysis will identify any number as candidate hash function with candidate hash algorithm A. We use this as a clue to determine if a hash function has been inlined. Specifically, if we have *duplicate candidate algorithms* in the target binary executable (multiple candidate hash functions with the same candidate algorithm), we assume the hash functions have been inlined and remove these when building the set of isolated hash functions.

The remaining candidate hash functions, that have a unique candidate algorithm in the target binary, are denoted *lone hash functions*. Formally, let $F_c(b, A)$ denote the set of candidate hash functions in binary executable b with candidate algorithm A. A candidate hash function f_{c_A} is a lone hash function if $|F_c(b, A)| = 1$.

6.1.2 Template-match Size Difference. The next clue we use for detecting inlined hash algorithms is the difference in the number of basic blocks in the CFG of the candidate hash function and the template-matches. We refer to this as the *template-match size difference*. We define the template-match size difference of a candidate hash function f_{c_A} as

$$\mathrm{DIFF}(f_{c_A}) = |V(\mathrm{CFG}(f_{c_A})| - |V(C_{f_{c_A}, i})|$$

where $C_{f_{c_A}, i}$ is the template-match of $\mathrm{CFG}(f_{c_A})$ with the maximum number of nodes. The intuition here is that if the template-match only makes up a small portion of the candidate hash function's CFG, the hash function is more likely to have been inlined in a larger function. A candidate hash function f_{c_A} is an *isolated hash function* f_{i_A} if

$$\mathrm{DIFF}(f_{c_A}) < K \wedge |F_c(b, A)| = 1.$$

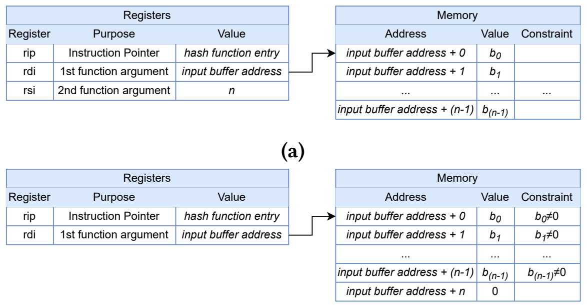
Through manual analysis and experimentation, we have discovered that using $K = 6$ provides a good balance between correctly identifying isolated hash functions and *Harm-DoS*'s execution time. This decision is discussed more in Section 10.

6.2 Symbolic Hash Function Analyses

We perform four symbolic execution tests on the isolated hash functions to build a profile of the behavior in order to determine if a nondisruptive patch can be made. The behavior profile is built with regard to signature (**MR6**), input-output relationships (**DR1**, **MR5**), case sensitivity (**MR7**), and memory accesses (**MR3**, **MR4**), of each.

Let $f(s_0)$ denote the set of program states created when performing symbolic execution, starting at function f with symbolic program state s_0. Let $\mathrm{FIN}(f(s_0))$ be the set of program states that reach a `ret` instruction of f, the *final states*. Let $\mathrm{WR}(f(s_0))$ be a set of tuples $\{(a_1, v_1), \ldots\}$, denoting the memory write operations that occur during symbolic execution. Each tuple consists of a symbolic expression for the address a_i and the value v_i of the memory write operation. Let $\mathrm{RD}(f(s_0))$ be defined similarly, but for memory read operations. For a symbolic expression e, $\mathrm{VAR}(e)$ denotes the symbolic variables used in the expression. With $\mathrm{MEM}(s, a)$, we denote the content of the memory at program state s at the address represented by symbolic expression a. Similarly, $\mathrm{REG}(s, r)$ denotes the content of register r. Let $\mathrm{ARG}(s_0, i)$ denote the value of the register, or memory location, that corresponds to the i-th argument of a function. For our purposes, we assume $\mathrm{ARG}(s_0, 1) = \mathrm{REG}(s_0, \texttt{rdi})$ and $\mathrm{ARG}(s_0, 2) = \mathrm{REG}(s_0, \texttt{rsi})$.

6.2.1 Symbolic Signature Detection. In Section 2.1 we explained that hash functions frequently implement one of two signatures, the buffer-length signature and buffer-only signature. We restrict



Registers		
Register	Purpose	Value
rip	Instruction Pointer	*hash function entry*
rdi	1st function argument	*input buffer address*
rsi	2nd function argument	*n*

Memory		
Address	Value	Constraint
input buffer address + 0	b_0	
input buffer address + 1	b_1	
...	...	...
input buffer address + (n-1)	$b_{(n-1)}$	

(a)

Registers		
Register	Purpose	Value
rip	Instruction Pointer	*hash function entry*
rdi	1st function argument	*input buffer address*

Memory		
Address	Value	Constraint
input buffer address + 0	b_0	$b_0 \neq 0$
input buffer address + 1	b_1	$b_1 \neq 0$
...	...	...
input buffer address + (n-1)	$b_{(n-1)}$	$b_{(n-1)} \neq 0$
input buffer address + n	0	

(b)

Figure 2: The symbolic starting state that we use for symbolic execution while determining if a function implements the (a) buffer-length and (b) buffer-only signature.

our focus to patching hash functions with these signatures, towards fulfilling **MR6**.

The main difference between the buffer-length and buffer-only signature is in how the end of the input buffer is determined. For the buffer-length signature, the length of the buffer is given explicitly as function argument. On the other hand, for the buffer-only signature, the end of the buffer is usually identified by a special byte, often a null byte.

For each signature, we set up a symbolic program state s_0 that corresponds to calling the hash function with symbolic input according to the signature. For the buffer-length signature, we create s_0 such that $\text{ARG}(s_0, 1) = x$, $\text{ARG}(s_0, 2) = n$. Here, x is a symbolic variable denoting a memory address of the input buffer and n is a defined, concrete length. Similarly, for the buffer-only signature, we create s_0 such that $\text{ARG}(s_0, 1) = x$, $\text{MEM}(s_0, x + i) = (b_i \neq 0)$ for $0 \leq i < n$ and $\text{MEM}(s_0, x + n) = 0$. Note, we assume the end of the buffer is indicated with a null byte. Figures 2a and 2b show a visual representation.

For each of the two signatures, we define a set of addresses $\mathcal{A}_\text{e}$ we expect to be accessed if the isolated hash function indeed implements the signature. For the buffer-length signature, $\mathcal{A}_\text{e} = \{x + i \mid 0 \leq i < n\}$, while for the buffer-only signature $\mathcal{A}_\text{e} = \{x + i \mid 0 \leq i \leq n\}$. Note that the number of memory addresses accessed for the buffer-only signature is one more than those for the buffer-length signature, because an additional address needs to be read in order to identify the end of the buffer. We determine which (if either) signature f implements by checking if we observe a memory read operation accessing each of the expected addresses during symbolic execution. That is, we calculate the following for each $\mathcal{A}_\text{e}$,

$$\{a_i \mid (a_i, v_i) \in \text{RD}(f(s_0)) \text{ and } x \in \text{VAR}(a_i)\} = \mathcal{A}_\text{e}.$$

Knowing which signature the isolated hash function implements provides us with insight into how the hash function receives its input. The benefit of this is twofold. It allows us to ensure that our replacement hash function consumes its input in the same way as the original, necessary for **MR6**, and it allows us to perform symbolic execution on the hash function with controlled input. We know which memory locations will be accessed, so we assign concrete values here, corresponding to the input. Let $\text{IN}(f, w)$ denote the program state s_0 set up in a way to perform symbolic execution on f with input buffer $w = b_0, b_1 \ldots b_{n-1}$. That is, we set up s_0 according to the signature of f and let $\forall i \in \{0..n-1\}$, $\text{MEM}(s_0, x+i) = b_i$. For a final state s_f let $\text{OUT}(s_f)$ denote the expression (symbolic or concrete) corresponding to the return value. For our purposes, we assume $\text{OUT}(s_f) = \text{REG}(s_f, \texttt{rax})$.

If the isolated hash function implements either the buffer-length signature, or buffer-only signature, we say the hash function is *signature-conforming*. The signature-conforming hash functions are passed to the following symbolic execution tests. Otherwise, the candidate hash function will not be patched.

6.2.2 *Symbolic Input-Output Matching.*

Each hash algorithm deterministically produces the output value, the *defined hash value*, for every given input. Since this relationship is unique to a specific algorithm, this can be used to identify the algorithm, as proposed by Gröbert [26, 33]. We include information on such input-output relationships in the detection model for each known-weak hash algorithm. In practice, we have observed small changes in the implementation of a hash algorithm that produce changes in the hash value. For this reason, we decide to associate a small set of defined hash values for each input buffer and hash algorithm. The first change we allow for, is hash functions producing 32-bit integer hash values versus 64-bit integer hash values. We also allow for two known variations in the source code implementation of the DJB hash function, which we show in Appendix C. Let $\mathcal{W}_\text{io} = \{(w_1, H_1), \ldots, (w_N, H_N)\}$. Each tuple (w_i, H_i) is an input buffer w_i and the set of defined hash values H_i, for the algorithm A. For example, for SDBM we have $(w_{i'}, H_{i'}) = (\texttt{abc}, \{97, 417419622498\})$ to associate input buffer abc with the corresponding 32-bit and 64-bit defined hash values, respectively.

To determine if a signature-conforming hash function implements the candidate algorithm, we use symbolic execution to obtain the *observed hash value* for given input. A set of observed hash values that each appear in the corresponding set of defined hash values, indicates that the signature-conforming hash function indeed implements the candidate algorithm, fulfilling **DR1**. Formally, f is *input-output conforming* (and therefore a *confirmed* hash function) with respect to A if

$$\forall (w_i, H_i) \in \mathcal{W}_\text{io},\ \forall s_f \in \text{FIN}(f(\text{IN}(f, w_i))),$$
$$|\text{FIN}(f(\text{IN}(f, w_i)))| = 1\ \wedge \text{OUT}(s_f) \in H_i$$

When choosing the input buffers to use for symbolic input-output matching, we create 256 buckets in the range $\{0..2^{32}\}$. For each known-weak hash algorithm, we choose input buffers that yield a defined hash value in each bucket. Additionally, we also choose 5 input buffers for which the algorithm outputs large defined hash values, approaching 2^{32}. By selecting inputs that correspond to large defined hash values, we can detect cases where the signature-conforming hash function yields output in a smaller range than 2^{32}. We cannot patch these while fulfilling **MR5**.

Finally, if we observe a symbolic hash value when evaluating the observed hash value, it means the concrete input we supplied was not sufficient to constrain the observed hash value to a single number. We mark such hash functions as not input-output-conforming. In all cases, functions that are not input-output conforming will not be patched.

6.2.3 Symbolic Case Sensitivity Checking. To determine if a signature-conforming hash function is case-sensitive, we provide it with pairs of concrete input buffers $\mathcal{W}_{\text{case}} = \{(w_{1,1}, w_{1,2}), \ldots, (w_{N,1}, w_{N,2})\}$. Each tuple $(w_{i,1}, w_{i,2})$ is a pair of input buffers which differ only in case, for example (abc, ABC). We say a signature-conforming hash function f is case-sensitive if two such buffers produce unequal hash values. Formally,

$$\exists (w_{i',1}, w_{i',2}) \in \mathcal{W}_{\text{case}} \mid \exists (s_{f,1}, s_{f,2}) \in \text{FIN}(f(\text{IN}(f, w_{i',1}))) \times \text{FIN}(f(\text{IN}(f, w_{i',2}))) \mid \text{OUT}(s_{f,1}) \neq \text{OUT}(s_{f,2})$$

The results of this analysis are stored to be used when constructing the replacement hash function, in order to fulfill **MR7**.

6.2.4 Symbolic Memory Access Analysis. It is important to know of any side effects of the signature-conforming hash function. Similar to the Hash Function Template-matching, described in Section 5.1, the purpose of this analysis is to identify side effects, to satisfy **MR3**. This is a more accurate, but slower analysis using symbolic execution.

Using symbolic execution, we can identify memory modifications, beyond the function's stack memory. If we observe such modifications, we decide not to patch the hash function. Otherwise, we refer to the isolated hash function as *memory-conforming*. This fulfills **MR3**. We set up s_0 according to the signature of f and create a symbolic variable x_r for each stack register $r \in \mathcal{R}_{\text{stack}}$. Then, we set $\text{REG}(s_0, r) = x_r$ for each $r \in \mathcal{R}_{\text{stack}}$. Formally, we say f is memory-conforming if

$$\forall\, (a_i, v_i) \in \text{WR}(f(s_0)),\ \text{VAR}(a_i) \subseteq \{x_r : r \in \mathcal{R}_{\text{stack}}\}.$$

If a hash function is signature-conforming, input-output-conforming, and memory-conforming, we say it is a *conforming hash function* and it is eligible for patching.

7 HASH TRANSPLANT

To perform the hash transplant, we construct a secure hash function, the *replacement hash function* and use it to replace the conforming hash function, the *original hash function.*

7.1 Replacement Hash Function Construction

Using the profile built during Pre-patch Examination, we construct a replacement hash function that is similar to the original in terms of signature and case sensitivity, fulfilling **MR6** and **MR7**.

We also add instructions at the start and end of the replacement hash function to store and restore the values of all registers that are modified in the replacement hash function, except the register housing the hash value. This is done to fulfill **MR2**. To avoid side effects and fulfill **MR4**, the replacement hash function is constructed to access only its input buffer and stack memory. Next, we discuss the two hash algorithms we use as a secure alternative.

7.1.1 Candidate Replacement Hash Algorithms. The first hash algorithm we use as replacement is SipHash, designed by Aumasson [3] specifically for the purpose of preventing hash-collision DoS attacks. SipHash achieves collision resistance by incorporating a 16-byte secret into its calculations. As long as this secret is unknown to an attacker, collisions can only be achieved through guessing [3], fulfilling **MR8**. The downside of SipHash is that it takes approximately 670 bytes to implement in AMD64 binary code. This makes it challenging to use SipHash as replacement hash algorithm, while fulfilling **MR1**, explained in Section 7.2. When the implementation size of SipHash makes it an impractical replacement algorithm, we use an alternative algorithm.

For this, we turn to strongly universal hash algorithms, introduced by Wegman [46]. A set of hash algorithms is strongly universal if, for a random hash algorithm, any input is mapped to every hash value with equal probability. If the attacker does not know which hash algorithm in the set is used, they cannot predictably generate collisions, as noted by Crosby [12]. Lemire introduces a strongly universal set of hash algorithms, named *Multilinear* [32]. This set of hash algorithms achieves collision resistance through a secret initial state of a pseudorandom number generator (PRNG). Therefore, this hash function is secure and can be used to fulfill **MR8**. We provide more details about our Multilinear hash algorithm approach in Appendix D.

The security guarantees of strongly universal hash algorithms are slightly weaker than those of SipHash. If an attacker can obtain a number of input-output pairs of the strongly universal hash algorithm, it is possible to calculate some of the random values used, mentioned by Aumasson [3]. This, in turn, will allow the attacker to pinpoint the specific hash algorithm used and therefore generate collisions. Therefore, we first try to replace a weak hash function with SipHash and only resort to using a Multilinear hash algorithm implementation if we fail to insert SipHash due to its implementation size.

Finally, an important decision to make is whether to randomize the secret used for the replacement hash function once when the weak hash function is patched, or every time the program is run. The latter may be more secure, as the attacker will not learn the secret by obtaining the patched executable. However, we have observed programs that store the key-value pairs produced by the hash function to a file [6]. Randomizing the hash function between these runs may break the functionality. Therefore, we have decided to limit randomization to the time when the binary executable is patched. The user can re-randomize the secret by patching the binary executable again.

7.2 Replacing the Hash Function

To patch the vulnerable binary executable, we leverage a process called *binary rewriting*. From a high level, we replace the original hash function by overwriting it with the replacement hash function. If the replacement hash function is larger than the original (in the number of bytes it requires to implement), we insert the remaining instructions, the *overflowing instructions*, elsewhere into the binary executable and adjust control flow so that these instructions are executed in the correct order.

Binary rewriting is complicated by the fact that many binary instructions, such as jump and data reference instructions, rely on their relative position in the binary. Simply inserting additional instructions in between existing instructions may break these relations, rendering the program defective. For this reason, it is safer to add instructions to a binary executable by overwriting others. To this end, we build on a method often used to modify binary executables, for example used by Bruschi [10] and Menon [37], namely we search for bytes in the binary executable that are never executed, colloquially called code caves. Code caves are often introduced by the compiler to align functions [22]. If no single code cave exists that is large enough to hold all the overflowing instructions, we join multiple caves with jump instructions. We show a full example of how to replace a hash function in Appendix B. In some cases, the binary executable does not contain enough code caves to house all the overflowing instructions. In these cases the patch fails and we report that there was no room for the replacement hash function. We discuss our decision to use this approach towards patching, as opposed to other existing approaches, in Section 12.

As we are only overwriting the instructions of the original hash function and, possibly, some instructions that serve only as padding bytes, we replace the original hash function while fulfilling **MR1**.

8 POST-PATCH EXAMINATION

After patching, we need to confirm that the replacement hash function does not introduce errors. Since proving equivalence of two programs is undecidable, we perform a local verification, aided by symbolic execution. We perform two steps, Symbolic Patch Memory Access Analysis and Symbolic Preimage Calculation. If either of these two steps fail, we say that replacing the hash function introduced errors into the program and discard the patch. Otherwise, the patch is considered a success.

8.1 Symbolic Patch Memory Access Analysis

Similar to the original hash function f_o, the replacement hash function f_r should not have any side effects. Moreover, f_r should access exactly the same bytes of the input buffer as f_o. To this end, we use the set of expected addresses $\mathcal{A}_\mathrm{e}$, defined in Section 6.2.4, for the signature of f_o. We perform symbolic execution on f_r and monitor the memory accessed in order to determine if the replacement was successful. We set up s_0 according to the signature of f_o and create a symbolic variable v_r for each stack register $r \in \mathcal{R}_{\text{stack}}$. Then, we set $\mathrm{REG}(s_0, r) = v_r$ for each $r \in \mathcal{R}_{\text{stack}}$. Let the address of the input buffer of be a symbolic variable x. We require that f_r only writes to stack memory:

$$\forall\ (a_i, v_i) \in \mathrm{WR}(f_\mathrm{r}(s_0)),\ \mathrm{VAR}(a_i) \subseteq \{v_r \mid r \in \mathcal{R}_{\text{stack}}\}$$

and reads from all the offsets of x appropriate for its signature:

$$\{a_i \mid (v_i, a_i) \in \mathrm{RD}(f_\mathrm{r}(s_0)) \text{ and } x \in \mathrm{VAR}(a_i)\} = \mathcal{A}_\mathrm{e}$$

and does not read from global memory:

$$\forall\ (a_i, v_i) \in \mathrm{RD}(f_\mathrm{r}(s_0)),\ \mathrm{VAR}(a_i) \subseteq (\{x\} \cup \{v_r \mid r \in \mathcal{R}_{\text{stack}}\}).$$

This fulfills **MR4**.

8.2 Symbolic Preimage Calculation

The purpose for this verification step is to ensure the replacement hash function only returns hash values that are within the range of the original hash function. We calculate preimages of the original hash function for hash values returned from the replacement hash function. We do this by evaluating the concrete hash value for the replacement hash function for a number of concrete inputs. For each of these concrete hash values, we use symbolic execution and constraint solving to obtain input for the original hash function for which it returns the same hash value. Note that such an evaluation is feasible due to the simple nature of known-weak hash algorithms. Formally, $\mathcal{W}_\mathrm{p} = \{w_1, w_2, \dots, w_N\}$ be a set of input buffers. We require,

$$\begin{aligned}&\forall w_i \in \mathcal{W}_\mathrm{p},\ \exists\, w_i' \mid \\ &\quad \forall (s_{f,\mathrm{o}}, s_{f,\mathrm{r}}) \in \mathrm{FIN}(f_\mathrm{o}(\mathrm{IN}(f_\mathrm{o}, w_i))) \times \mathrm{FIN}(f_\mathrm{r}(\mathrm{IN}(f_\mathrm{r}, w_i'))), \\ &\qquad\qquad \mathrm{OUT}(s_{f,\mathrm{o}}) = \mathrm{OUT}(s_{f,\mathrm{r}})\end{aligned}$$

We do this towards fulfilling **MR5**.

9 IMPLEMENTATION

Harm-DoS is implemented in approximately 3,500 lines of Python code in an open-source repository[3]. It leverages the angr binary program analysis framework [2] for the fundamental requirements of binary program analysis, such as disassembly, CFG recovery and symbolic execution. All the analysis steps discussed in this paper are implemented by the authors, using only these fundamentals. As input, *Harm-DoS* takes the names of executable files to analyze and writes the analysis results for each to a file in JSON format. The patched executable is also produced, if applicable. In the current implementation, only ELF executable files for the AMD64 architecture are supported.

10 EXPERIMENTAL RESULTS

Our experimental evaluation is composed of three main parts. First, in Section 10.1 we analyze a large data set of real-world binaries to test the ability of *Harm-DoS* to detect and patch vulnerabilities at scale. Second, in Section 10.2, we constitute and analyze a subset of these binaries, containing manually identified hash functions. We use this to determine the accuracy of our approach based on a known ground truth. Third, we discuss a case study in Section 10.3. All experiments were run with PyPy 7.3.5, in an Ubuntu 20 Docker container with 10 CPU cores. Analyzing a single binary executable take approximately 215MB memory on average.

10.1 Full-Scale Analysis

In this section we present the results we have obtained by running *Harm-DoS* on a large set of real-world binaries. Our data set consists of 105,831 unique AMD64 ELF executable files, extracted from the AllStar data set [45]. This data set contains the binaries obtained when building the Jessie distribution of the Debian packages. These packages are built with the `debuild` tool which uses the compiler specified in the package configuration to create the executable. On average, these binaries are approximately 5.6MB large, consisting of approximately 58,000 basic blocks and 110,000 CFG edges. Among these packages are many widely used programs, such as Firefox, Apache, PHP and Binutils. Note that even though the binaries in this data set contain debugging symbols, *Harm-DoS* does not rely on these in any phase of analysis.

10.1.1 Discovery. We run Vulnerability Diagnosis on all of the binaries in the data set. This takes about 2 days to complete, with an average analysis time of 17s per binary. We identify 31,052 candidate hash functions in 8,930 binaries. Table 2 shows the number of

[3]https://github.com/usc-isi-bass/hashdos_vulnerability_detection

Table 2: The candidate, lone, isolated, confirmed and patched hash functions from the full-scale analysis.

Hash Alg	Candidate	Lone	Isolated	Confirmed	Patched
BKDR	11,053	2,568	45	2	2
DEK	2,944	1,872	976	0	0
DJB	2,558	1,724	424	390	372
ELF	2,069	1,318	433	321	317
FNV	3,327	141	82	46	44
JS	0	0	0	0	0
RS	69	21	0	0	0
SDBM	9,032	2,442	301	37	24
Total	31,052	10,086	2,261	796	759

candidate hash functions per candidate algorithm. The table shows that not a single candidate hash function with JS as candidate algorithm was discovered. To ensure this is not caused by false negative detections, we performed a cursory manual search of the data set, which also did not yield any such hash functions. The lack of JS hash functions, therefore seems to originate from a lack of their use in practice.

In the next phase of analysis, we reduce the set of candidate hash functions to those that are not inlined – the isolated hash functions. The first step of this analysis is to remove all candidate hash functions with duplicate candidate algorithms in the same binary. Table 2 shows that $10,086$ lone hash functions remain after this step.

The next step is to remove the lone hash functions for which the template-match size difference exceeds 6 basic blocks. We discuss this decision in Section 10.2. Table 2 shows the number of isolated hash functions, whose behavior *Harm-DoS* will analyze with symbolic execution.

10.1.2 Patching. In this section, we discuss how effectively we can patch the confirmed hash functions. Analyzing all the isolated hash functions takes about 3.5 days with an average analysis time of 28s per binary, showing that *Harm-DoS* is scalable. In Table 2 we show the patching results in the *Patched* column. Approximately 35% of the isolated hash functions were confirmed as weak hash functions and 95% of these confirmed hash functions were patched successfully. We use the ratio of successful patches over the confirmed hash functions, as this excludes all false positive identifications in Vulnerability Diagnosis and only includes true positive hash functions, verified with Symbolic Input-Output Matching. Note that this also excludes inlined hash functions which we cannot verify with symbolic execution. Recall that in order to calculate the observed output, a hash function must be signature-conforming. Therefore, if a hash function is either signature non-conforming or input-output non-conforming, it cannot be confirmed as hash function. In Table 3 we show how often these two reasons lead to an isolated hash function not being confirmed as hash function. In the vast majority of cases, signature-nonconformity prevented the patch. Through manual analysis we verified that most cases of non-conforming signature were indeed not hash functions, but instead included other functions that share popular constants with a weak hash function (and are thus flagged as candidates during optimistic weak hash function discovery in the Vulnerability Diagnosis phase). Thus our approach correctly filters out these misidentified candidates in the symbolic execution analysis. For the DEK hash algorithm in particular, we observe that in 99% of these cases a portion of the MD5 hash algorithm is misclassified as DEK hash, because it includes the constant-mnemonic pair match (5, `{rol}`).

Table 3: The number of times signature-nonconformity and input-output nonconformity lead to an isolated hash function not being confirmed. The right-most column shows when analysis time exceeded 4 hours or an error occurred. The large number of signature-nonconforming isolated hash functions is caused by too-broad hash function detection in the optimistic static analysis in the Vulnerability Diagnosis phase. These false positives are correctly filtered out during symbolic execution.

Hash Alg	Non-confirmed	sig	IO	TOs and Errors
BKDR	43	40	3	0
DEK	976	865	37	74
DJB	34	25	5	4
ELF	112	47	52	13
FNV	36	5	31	0
SDBM	264	90	54	120
Total	1465	1,072	182	211

In Section 7.1.1 we explain that SipHash has stronger security properties than the Multilinear hash function and is therefore the preferred replacement hash algorithm. From our experiments, we found that of the 759 hash functions that were patched successfully, SipHash was used as the replacement 702 times, or 92.49%.

10.2 Ground Truth Analysis

In order to measure how well *Harm-DoS* detects all known-weak hash functions, we construct ground truth data set. We select 202 hash functions, for which we have manually verified from the source code that they implement a given known weak hash algorithm. We draw our functions from a subset of 156 binaries used in our full-scale analysis of the AllStar data set [45]. We attempted to balance representation of different known weak hash algorithms in our ground truth data set, but in reality some algorithms were much more frequent in real-world programs than others. We show the number of manually identified hash functions, included the ground truth data set, in the MI column of Table 4.

For the Vulnerability Diagnosis phase, we show the number of true positive classifications in Table 4. A true positive is a hash function that is correctly detected by our analysis as a candidate hash function, with the correct candidate algorithm. Conversely, a false negative is a hash function either not detected by our analysis, or is paired with an incorrect candidate algorithm (i.e., mistaken for another algorithm). Note that in Table 4 for each algorithm the number of candidate hash functions match the number of manually identified hash functions. Therefore, *Harm-DoS* detected all hash functions correctly.

Next, we look at how well *Harm-DoS* detects inlined true positive hash functions. In Table 4, we show the number of isolated hash functions. To measure the accuracy of the Filtering Inlined Hash Functions analysis (Section 6.1) we perform a manual inspection of

Table 4: The results from running *Harm-DoS* on the binaries in the ground truth data set. The *MI* column shows the number of manually identified hash functions. The *Candidate* column shows how many manually identified hash functions were identified as candidate hash functions, i.e. true positive classifications. The *Isolated* column shows how many of these remain after filtering out the inlined functions.

Hash Alg	MI	Candidate	Isolated
BKDR	1	1	1
DJB	50	50	34
ELF	50	50	30
FNV	50	50	9
RS	8	8	0
SDBM	43	43	33
Total	202	202	107

the candidate hash functions that are removed. The results of this inspection is shown in Table 5. We place each of these candidate hash function into one of four categories. ***Cat1*:** the target candidate hash function is indeed inlined. ***Cat2*:** the target candidate hash function is not inlined, but there are inlined, duplicate candidate hash algorithms. ***Cat3*:** the target candidate hash function is not inlined and none of the duplicate candidate hash algorithms are inlined. ***Cat4*:** the target candidate hash function is not inlined and it is a lone hash function, but the template-match size difference is greater than 6. For categories *Cat1* and *Cat2*, the analysis correctly decided not to patch the candidate hash function. For *Cat2*, only replacing the target candidate hash function and not the rest, will break functionality (see **DR2**). For *Cat3*, since none of the duplicate candidate algorithm are inlined, we can replace each individually. However, distinguishing between *Cat2* and *Cat3* automatically is difficult and we leave this for future research. For *Cat4*, we observed an isolated hash function with a template-match size difference of up to 13, due to a loop-unrolled implementation of SDBM. To investigate this further, we reran the full-scale analysis with an increased template-match size difference limit of 13. We observed that in this case *Cat4* is empty, however no additional hash functions were patched successfully. For the SDBM hash function mentioned above, there was a failure in the post-patch examination. Therefore, these cannot be patched automatically while guaranteeing correctness with respect to **MR5**. We have also observed RS hash functions with template-match size difference 8. These hash functions are added to the binary executable by a Pascal compiler [23] for use in a hash table. The hash function is implemented in a specialized way to compute a hash value for Pascal string objects. These objects are passed as a pointer to the hash function. The string length and data are accessed via constant offsets of this pointer. Note that this does not match either the buffer-length or buffer-only signature. Consequently, it does not follow the context-agnostic nature of the hash functions described in Section 2.1 and cannot be patched automatically while ensuring correctness. For this reason, we decide to use a template-match size difference of 6 allowing, us to capitalize on both analysis time and patchability.

Table 5: The results from manually inspecting the candidate hash functions that are not isolated hash functions.

Hash Alg	Removed	Cat1	Cat2	Cat3	Cat4
DJB	16	5	7	1	3
ELF	20	5	9	6	0
FNV	41	0	39	2	0
RS	8	0	0	0	8
SDBM	10	0	0	4	6
Total	95	10	55	13	17

10.2.1 Test Case Verification. As an extra layer of verification, we use the test cases included with some Debian packages to confirm that *Harm-DoS* does not break functionality. These test cases are provided via the GNU Make build utility [20]. We identify and run the test cases of 21 packages with binaries patched by *Harm-DoS*. When necessary, we also rebuild hash tables stored in files, used by the test cases, with the patched binaries. In every case, the patched binaries do not introduce test failures.

10.3 Case Study

To show the effectiveness of *Harm-DoS* in a real-world context, we discuss a remotely-exploitable zero-day hash-collision vulnerability that we discovered, and patched, in Snudown [44], a component of Reddit [41]. We disclosed this vulnerability to Reddit in accordance to the coordinated disclosure policy. The vulnerability was assigned ID CVE-2021-41168 [16]. We also implemented a mitigation that replaces the weak hash function with SipHash, which was accepted by the developers.

Snudown is a library used in Reddit to convert markdown to HTML. This library uses a hash table to map reference labels to their links, using the SDBM hash algorithm.

We launch a proof-of-concept attack against Snudown, running locally, by parsing a large number of references with labels crafted to cause collisions in the hash table. We measure the parsing time of an increasing number of colliding reference labels. As a sanity check, we repeat the experiment with random labels. We plot the parsing time against the input size in Figure 3. The significant difference in parsing time growth between the malicious and random labels confirms the vulnerability. Note, the superlinear growth in parsing time for random labels, is caused by coincidental collisions due to the small table size.

We run *Harm-DoS* on the executable, which successfully produces a patched Snudown. To show the malicious growth in parsing time no longer occurs, we relaunch the same attack on the patched executable, shown in Figure 3. It is clear that patched Snudown does not suffer from the same vulnerability. Moreover, due to the secret used in patched Snudown, the attacker cannot adapt the attack to reliably trigger collisions. To show that we have mitigated the vulnerability without introducing errors, we run the test cases that are supplied with the Snudown project, which all pass. This shows that *Harm-DoS* can be used to identify and mitigate real-world vulnerabilities.

Note, in Figure 3 parsing time is less for malicious references, parsed with patched Snudown. This is unrelated to the hash function, since both the malicious and random labels are distributed

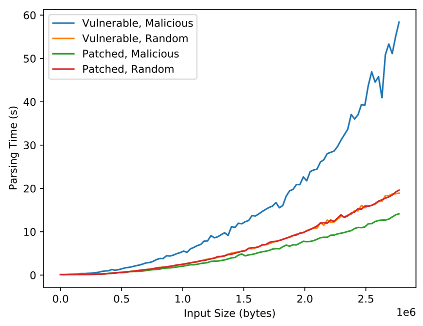


Figure 3: Parsing time for both vulnerable and patched Snudown for both malicious and random reference labels.

evenly across the hash table. The difference in parsing time originates from CPU caching, leading to a slower time per iteration when searching the list of coincidental collisions.

11 LIMITATIONS AND FUTURE WORK

In this section, we discuss the current limitations of *Harm-DoS* and future work to improve it.

Patching Inlined Hash Functions. As motivated by **MR1**, it is necessary to determine exactly which instructions implement the weak hash algorithm. For inlined hash functions, this requires *Harm-DoS* to distinguish between the instructions implementing the hash algorithm and those of the caller function. A possible approach to this end is to use data-flow analysis. This can be used to compute a program slice over the instructions that form part of the hash algorithm. The start and end of the hash algorithm in this slice can be identified by using Symbolic Input Output matching (Section 6.2.1) on various subexpressions extracted via symbolic execution. However, due to the computationally expensive nature of symbolic execution, it is unclear whether such an approach will be feasible.

Reassembleable Disassembly. In Section 10.1 we have shown that in 57 cases SipHash could not be used to mitigate the vulnerability, because of its implementation size. In 13 cases, the vulnerability could not be patched at all. Recent advancements in *reassembleable disassembly* could be useful here, made by Flores-Montoya [19] and Bauman[5], as they allow for arbitrary changes to the disassembled code, including introducing new functions. Note however that such approaches are subject to limitations in terms of trade-offs between scalability and correctness. This is achieved by inferring the symbols used during compilation and adding these to the disassembled code.

12 RELATED WORK

Hash Function Discovery. Lestringant defined an approach to detect implementations of cryptographic algorithms [33]. It searches for the unique constants, similar to *Harm-DoS*. The scope of the paper, however ends at detecting the cryptographic algorithm and the analysis thereof is left to a human expert. *Harm-DoS*, on the other hand, requires an analysis that is capable of determining if the identified hash function is patchable.

Another detection mechanism for cryptographic algorithms, mentioned by Lestringant [33] and Gröbert [26], is to compare the output for corresponding input to subcomputations of the algorithm. We use the same approach Symbolic Input-Output Matching (Section 6.2.2). In our case, however, the hash functions are simple enough that we can calculate the output of the entire function for over 200 inputs.

Other code similarity approaches often rely on creating a message digest (MD) of the code, as done by Farhadi [18], Xu [47] and Ghidra [24], as well as testing for CFG subgraph isomorphism as done by Bruschi [10]. To test the feasibility of detecting weak hash functions using MDs, we used the Function ID feature of Ghidra [24]. This feature creates a MD of the function using its sequence of instructions, including the mnemonic, register names, memory accesses and constants. We created a MD for each hash function used to create the detection models used by *Harm-DoS*. However, this approach failed to detect any hash function in the ground truth data set used in our evaluation. This is to be expected, as MDs are used to identify *syntactically* equivalent code. Consequently, trivial changes made to the hash function by the compiler (e.g. instruction order) will yield a different MD. This approach is therefore less well suited for our purposes than our approach, which relies on features inherent to weak hash functions, discussed in Section 2.1. We expect other MD based approaches, such as the IDA FLIRT [28] approach to perform similarly.

We also measured the feasibility of using CFG subgraph isomorphism to detect weak hash functions. We observed this approach is very susceptible to failure, caused by slight modifications made to the CFG by the compiler. If *Harm-DoS* used subgraph isomorphism it would fail to detect 13 of the 202 hash functions in our ground truth data set, while our approach correctly identified all of these hash functions. Moreover, due to the computational complexity of subgraph isomorphism, the analysis takes approximately 3 hours to process the 156 binaries in the ground truth data set, where our approach takes about 40 minutes. Therefore, *Harm-DoS* is more suitable for analysing a large number of binaries.

Meijer introduces an approach to detect previously unseen cryptographic algorithms [35] by defining signatures to use for subgraph isomorphism in the data-flow graph (DFG). This approach is not well-suited for detecting weak hash functions because these have very simple data flow, as discussed in Section 2.1, making it prone to many false positives.

Petsios and Blair, respectively propose fuzzing based approaches for detecting algorithmic complexity vulnerabilities [9, 39]. Given that fuzzing is a randomized dynamic analysis, using such an approach is inherently slower than using a lightweight static analysis. Moreover, in both these approaches automatic patching of the vulnerabilities are out of scope.

Mitigation Strategies. Hash-collision vulnerabilities can also be mitigated by limiting the number of collisions, as done in DJBDNS [7] for cached DNS queries. This solution is unsatisfactory, since additional colliding entries are discarded, even if they originate from a benign user. At the network level it is possible to terminate malicious network connections, as done by Meng [36].

The downside is that an entire additional system is required to prevent exploitation. In contrast, *Harm-DoS* addresses the root cause of the vulnerability, the weak hash function.

Hash Function Patching. Bruschi and Menon, respectively achieve binary rewriting by repurposing unused bytes [10, 37]. In the former, sequences of no operation (nop) instructions are overwritten to insert malicious code. This malicious code executes independently from the host executable and therefore it is unnecessary to take the context in which it is placed into consideration. In the latter, unused alignment bytes are used to patch buffer-overflow vulnerabilities by exiting the program gracefully on malicious input. *Harm-DoS*, on the other hand, keeps the program running normally.

Other approaches, such as used by Duck [17], rely on dynamically mapped memory to insert patch code. This memory is mapped during program execution and the patch code is loaded from disk into it. Consequently, the patch code is not available for static analysis. In this regard, *Harm-DoS* is better as it inserts the patch code directly into the executable file, making it available for static analysis to ensure correctness.

13 CONCLUSION

We have presented a novel approach for automatically detecting and replacing weak hash functions in binary code in order to prevent algorithmic complexity vulnerabilities. We evaluated our prototype on a large data set of 105, 831 real-world binaries, identified 796 confirmed weak hash functions, and successfully replaced 759 of these in a non-disruptive manner. We show that *Harm-DoS* is scalable, evident by our processing a large data set of 100 K binaries in about 5.5 days. *Harm-DoS* is also effective in a real-world context – we used it to discover a zero-day vulnerability in Reddit.

ACKNOWLEDGMENTS

This material is based on research sponsored by the National Science Foundation via contract number 1815495. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation.

REFERENCES

[1] Jyrki Alakuijala, Bill Cox, and Jan Wassenberg. 2016. Fast keyed hash/pseudo-random function using SIMD multiply and permute. *CoRR* abs/1612.06257 (2016). arXiv:1612.06257 http://arxiv.org/abs/1612.06257

[2] angr. 2016. The Angr binary analysis platform. http://angr.io.

[3] Jean-Philippe Aumasson and Daniel J. Bernstein. 2012. SipHash: A Fast Short-Input PRF. In *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7668)*, Steven D. Galbraith and Mridul Nandi (Eds.). Springer, 489–508. https://doi.org/10.1007/978-3-642-34931-7_28

[4] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 845–860. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao

[5] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_05A-4_Bauman_paper.pdf

[6] Dan J. Bernstein. 2000. CDB. https://cr.yp.to/cdb.html.

[7] Dan J. Bernstein. 2001. DJBDNS. https://cr.yp.to/djbdns.html.

[8] Dan J. Bernstein. 2003. DJB Hash. http://www.cse.yorku.ca/~oz/hash.html.

[9] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2020. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/hotfuzz-discovering-algorithmic-denial-of-service-vulnerabilities-through-guided-micro-fuzzing/

[10] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4064)*, Roland Büschkes and Pavel Laskov (Eds.). Springer, 129–143. https://doi.org/10.1007/11790754_8

[11] Richard M. Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. 2009. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*. IEEE Computer Society, 186–199. https://doi.org/10.1109/CSF.2009.13

[12] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association. https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks

[13] CVE-2011-4885 2011. CVE-2011-4885. Available from CVE Details, CVE-ID CVE-2011-4885.. https://www.cvedetails.com/cve/CVE-2011-4885/

[14] CVE-2012-1150 2012. CVE-2012-1150. Available from National Vulnerability Database, CVE-ID CVE-2009-1897.. https://nvd.nist.gov/vuln/detail/CVE-2012-1150

[15] CVE-2012-2739 2012. CVE-2012-1150. Available from National Vulnerability Database, CVE-ID CVE-2012-2739.. https://nvd.nist.gov/vuln/detail/CVE-2012-2739

[16] CVE-2021-41168 2021. CVE-2021-41168. Available from National Vulnerability Database, CVE-ID CVE-2021-41168.. https://nvd.nist.gov/vuln/detail/CVE-2021-41168

[17] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 151–163. https://doi.org/10.1145/3385412.3385972

[18] Mohammad Reza Farhadi, Benjamin C. M. Fung, Philippe Charland, and Mourad Debbabi. 2014. BinClone: Detecting Code Clones in Malware. In *Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, California, USA, June 30 - July 2, 2014*. IEEE, 78–87. https://doi.org/10.1109/SERE.2014.21

[19] Antonio Flores-Montoya and Eric M. Schulte. 2020. Datalog Disassembly. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1075–1092. https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[20] GNU Project Free Software Foundation. 2022. GNU make. https://www.gnu.org/software/make/manual/make.html.

[21] Glenn Fowler, Phong Vo, and Landon Curt Noll. 2012. The FNV Non-Cryptographic Hash Algorithm. https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-03.

[22] Inc Free Software Foundation. 2022. Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html.

[23] freepascal 2021. Free Pascal. https://www.freepascal.org/.

[24] ghidra 2022. Ghidra. https://ghidra-sre.org/.

[25] gnu 2022. GCC, the GNU Compiler Collection. http://www.gnu.org/software/gcc/index.html.

[26] Felix Gröbert, Carsten Willems, and Thorsten Holz. 2011. Automated Identification of Cryptographic Primitives in Binary Programs. In *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6961)*, Robin Sommer, Davide Balzarotti, and Gregor Maier (Eds.). Springer, 41–60. https://doi.org/10.1007/978-3-642-23644-0_3

[27] Christian Heimes. [n.d.]. PEP 456 – Secure and interchangeable hash algorithm. https://www.python.org/dev/peps/pep-0456/.

[28] Hex-Rays. [n.d.]. IDA F.L.I.R.T. Technology: In-Depth. https://hex-rays.com/products/ida/tech/flirt/in_depth/.

[29] Brian Kernighan and Dennis Ritchie. 1972. *The C Programming Language*. Prentice Hall PTR, New Jersey, USA.

[30] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static Analysis for Regular Expression Denial-of-Service Attacks. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7873)*, Javier López, Xinyi Huang, and Ravi S. Sandhu (Eds.). Springer, 135–148. https://doi.org/10.1007/978-3-642-38631-2_11

[31] Donald Ervin Knuth. 1998. *The art of computer programming, , Volume III, 2nd Edition.* Addison-Wesley. http://www.worldcat.org/oclc/312994415

[32] Daniel Lemire and Owen Kaser. 2014. Strongly Universal String Hashing is Fast. *Comput. J.* 57, 11 (2014), 1624–1638. https://doi.org/10.1093/comjnl/bxt070

[33] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. 2015. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn (Eds.). ACM, 203–214. https://doi.org/10.1145/2714576.2714639

[34] George Marsaglia et al. 2003. Xorshift rngs. *Journal of Statistical Software* 8, 14 (2003), 1–6.

[35] Carlo Meijer, Veelasha Moonsamy, and Jos Wetzels. 2021. Where's Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 555–572. https://www.usenix.org/conference/usenixsecurity21/presentation/meijer

[36] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. 2018. Rampart: Protecting Web Applications from CPU-Exhaustion Denial-of-Service Attacks. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 393–410. https://www.usenix.org/conference/usenixsecurity18/presentation/meng

[37] Jayakrishna Menon, Christophe Hauser, Yan Shoshitaishvili, and Stephen Schwab. 2018. A Binary Analysis Approach to Retrofit Security in Input Parsing Routines. In *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*. IEEE Computer Society, 306–322. https://doi.org/10.1109/SPW.2018.00049

[38] perlsec - Perl security. 2003. Algorithmic Complexity Attacks. https://perldoc.perl.org/perlsec.html.

[39] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2155–2168. https://doi.org/10.1145/3133956.3134073

[40] M. V. Ramakrishna and Justin Zobel. 1997. Performance in Practice of String Hashing Functions. In *Database Systems for Advanced Applications '97, Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA), Melbourne, Australia, April 1-4, 1997 (Advanced Database Research and Development Series, Vol. 6)*, Rodney W. Topor and Katsumi Tanaka (Eds.). World Scientific, 215–224.

[41] reddit 2005. Reddit. https://www.reddit.com.

[42] sdbm 2007. SDBM Library. https://apr.apache.org/docs/apr-util/0.9/group__APR__Util__DBM__SDBM.html.

[43] Robert Sedgewick. 1990. *Algorithms in C.* Addison-Wesley Professional, Bosotn, MA.

[44] snudown 2018. Snudown. https://www.github.com/reddit/snudown.

[45] JHU/APL Staff. 2019. Assembled Labeled Library for Static Analysis Research (ALLSTAR) Dataset. http://allstar.jhuapl.edu/

[46] Mark N. Wegman and Larry Carter. 1981. New Hash Functions and Their Use in Authentication and Set Equality. *J. Comput. Syst. Sci.* 22, 3 (1981), 265–279. https://doi.org/10.1016/0022-0000(81)90033-7

[47] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 462–472. https://doi.org/10.1109/ICSE.2017.49

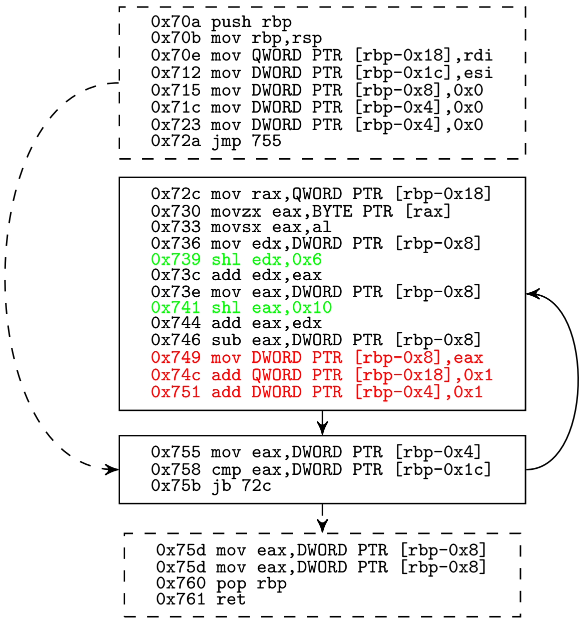
A VULNERABILITY DIAGNOSIS EXAMPLE

We illustrate the process of detecting weak hash functions in binary code with an example. Consider Figure 4, showing the assembly code, arranged in a CFG, for a hash function implementing SDBM. After disassembling the target binary executable and discovering the functions therein, the analysis considers each of these functions individually as target function.

The first step of *Harm-DoS* when analyzing a target function, is to determine if it matches the hash function template. To this end, it identifies the nontrivial SCCs in the CFG of the function (if any). The CFG shown in Figure 4 has a single nontrivial SCC, which we highlight with solid nodes and edges.

Next, *Harm-DoS* iterates through the instructions in the SCC and analyzes memory-write operations. In Figure 4 instructions that write to memory are identified at addresses `0x749`, `0x74c` and `0x751`. Each of these write to a memory address calculated as a constant offset from the stack base pointer register, `rbp`. As mentioned in Section 5.1, these are the only type of memory-write operations allowed. Therefore, this SCC passes and is marked as a template-match and *Harm-DoS* proceeds to perform Constant-Mnemonic Pair Discovery.

Since the target function is template-matching, *Harm-DoS* searches for the constant-mnemonic pair fingerprints of the detection model of each known-weak hash algorithm. When considering the constant-mnemonic pair fingerprints of SDBM, the analysis discovers the constants 6 and 16 (`0x10` in hexadecimal), each used with mnemonic `shl`. These can be seen at addresses `0x739` and `0x741` in Figure 4. As this matches the constant-mnemonic pair fingerprints for SDBM (see Table 1), the analysis notes the discovery of a constant-mnemonic pair match. Therefore, it identifies the target function as a candidate hash function with SDBM as candidate algorithm.



```
0x70a push rbp
0x70b mov rbp,rsp
0x70e mov QWORD PTR [rbp-0x18],rdi
0x712 mov DWORD PTR [rbp-0x1c],esi
0x715 mov DWORD PTR [rbp-0x8],0x0
0x71c mov DWORD PTR [rbp-0x4],0x0
0x723 mov DWORD PTR [rbp-0x4],0x0
0x72a jmp 755

0x72c mov rax,QWORD PTR [rbp-0x18]
0x730 movzx eax,BYTE PTR [rax]
0x733 movsx eax,al
0x736 mov edx,DWORD PTR [rbp-0x8]
0x739 shl edx,0x6
0x73c add edx,eax
0x73e mov eax,DWORD PTR [rbp-0x8]
0x741 shl eax,0x10
0x744 add eax,edx
0x746 sub eax,DWORD PTR [rbp-0x8]
0x749 mov DWORD PTR [rbp-0x8],eax
0x74c add QWORD PTR [rbp-0x18],0x1
0x751 add DWORD PTR [rbp-0x4],0x1

0x755 mov eax,DWORD PTR [rbp-0x4]
0x758 cmp eax,DWORD PTR [rbp-0x1c]
0x75b jb 72c

0x75d mov eax,DWORD PTR [rbp-0x8]
0x75d mov eax,DWORD PTR [rbp-0x8]
0x760 pop rbp
0x761 ret
```

Figure 4: A CFG showing the basic blocks of an implementation of the SDBM hash algorithm. Solid nodes and edges indicate the template-match.

B HASH TRANSPLANT EXAMPLE

To illustrate the hash transplant procedure, we use a replacement hash function from the Multilinear set of universal hash functions to replace the hash function shown in Listing 1. In order to generate pseudorandom values at run time, we add an implementation of the Xorshift PRNG [34], introduced by Marsaglia, to the replacement hash function. A source code representation of the replacement hash function is shown in Listing 2. The initialization values for the hash value (line 3) and the random state (line 4) are chosen randomly for every patch and are therefore unknown to an attacker. On line 8, the hash value is updated in a loop according to the next input character. Lines 9 to 11 implement the Xorshift PRNG.

Next, we illustrate the process of replacing a hash function, using the weak hash function shown in Listing 1, a source code implementation of the DEK hash algorithm. Note that in Listing 2 the replacement hash function receives its input and yields output in exactly the same way as the original hash function (line 1). The assembly code for the replacement hash function is shown in Listing 3. Figure 5 shows how the assembly code of the replacement hash function is inserted in the binary. The first five instructions of the replacement hash are inserted by overwriting the original hash function. The 16 overflowing instructions are inserted in 4 code caves. The grayed-out instructions represent the last instruction of the function before the code cave starts.

```
1  unsigned int hash(const char* str, unsigned int len) {
2     unsigned int h = len;
3     unsigned int i = 0;
4     for (i = 0; i < len; ++str, ++i) {
5        h = ((h << 5) ^ (h >> 27)) ^ (*str);
6     }
7     return h;
8  }
```

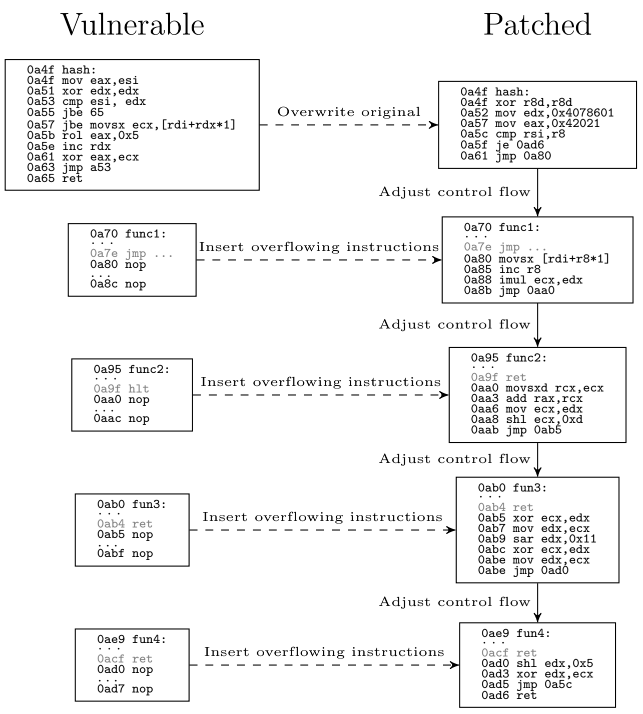
Listing 1: The hash function we aim to replace, a source code implementation of the DEK hash algorithm [31].

```
1   unsigned int univ_hash(const char* str, unsigned int len) {
2       char c;
3       int h = 270369;
4       int r = 67601921;
5       int i;
6       for (i = 0; i < len; i++) {
7           c = str[i];
8           h += (r * c);
9           r ^= r << 13;
10          r ^= r >> 17;
11          r ^= r << 5;
12      }
13      return h;
14  }
```

Listing 2: A source code representation of the hash function we use as a replacement to the weak hash function, Listing 1.

```
xor     r8d,r8d
mov     edx,0x4078601
mov     eax,0x42021
loop:
cmp     rsi,r8
je      done
movsx   ecx,[rdi+r8]
inc     r8
imul    ecx,edx
movsxd  rcx,ecx
add     rax,rcx
mov     ecx,edx
shl     ecx,0xd
xor     ecx,edx
mov     edx,ecx
sar     edx,0x11
xor     ecx,edx
mov     edx,ecx
shl     edx,0x5
xor     edx,ecx
jmp     loop
done:
ret
```

Listing 3: Assembly code of Listing 2.


Vulnerable
Patched
Overwrite original
Adjust control flow
Insert overflowing instructions
Adjust control flow
Insert overflowing instructions
Adjust control flow
Insert overflowing instructions
Adjust control flow
Insert overflowing instructions

```
0a4f hash:
0a4f mov eax,esi
0a51 xor edx,edx
0a53 cmp esi, edx
0a55 jbe 65
0a57 jbe movsx ecx,[rdi+rdx*1]
0a5b rol eax,0x5
0a5e inc rdx
0a61 xor eax,ecx
0a63 jmp a53
0a65 ret
```

```
0a4f hash:
0a4f xor r8d,r8d
0a52 mov edx,0x4078601
0a57 mov eax,0x42021
0a5c cmp rsi,r8
0a5f je 0ad6
0a61 jmp 0a80
```

```
0a70 func1:
. . .
0a7e jmp ...
0a80 nop
. . .
0a8c nop
```

```
0a70 func1:
. . .
0a7e jmp ...
0a80 movsx [rdi+r8*1]
0a85 inc r8
0a88 imul ecx,edx
0a8b jmp 0aa0
```

```
0a95 func2:
. . .
0a9f hlt
0aa0 nop
. . .
0aac nop
```

```
0a95 func2:
. . .
0a9f ret
0aa0 movsxd rcx,ecx
0aa3 add rax,rcx
0aa6 mov ecx,edx
0aa8 shl ecx,0xd
0aab jmp 0ab5
```

```
0ab0 fun3:
. . .
0ab4 ret
0ab5 nop
. . .
0abf nop
```

```
0ab0 fun3:
. . .
0ab4 ret
0ab5 xor ecx,edx
0ab7 mov edx,ecx
0ab9 sar edx,0x11
0abc xor ecx,edx
0abe mov edx,ecx
0abe jmp 0ad0
```

```
0ae9 fun4:
. . .
0acf ret
0ad0 nop
. . .
0ad7 nop
```

```
0ae9 fun4:
. . .
0acf ret
0ad0 shl edx,0x5
0ad3 xor edx,ecx
0ad5 jmp 0a5c
0ad6 ret
```

Figure 5: An illustration of how a hash function is replaced, by overwriting the original and adding instructions to code caves. Every code cave (on the left) receives a number of instructions (on the right) of the patch, shown in Listing 3. The grayed-out instructions show the last instruction of the function before the padding bytes (i.e. the code cave) starts. Every code cave ends with a jump instruction to the code cave housing the next patch instructions.

C DJB HASH VARIATIONS

We show two common variations of the DJB hash algorithm in Listings 4 and 5. In the source code implementation illustrated in Listing 4, often referred to as DJBX33A, an addition operation is performed between the current hash value and the next input byte. On the other hand, in Listing 5, often referred to as DJBX33X, an exclusive or operation is used instead.

```
1  uint djbx33a(char* str, uint len) {
2    uint h = 5381;
3    uint i = 0;
4    for (i=0; i<len; ++i) {
5      h=((h<<5)+h)+(str[i]);
6    }
7    return h;
8  }
```

Listing 4: One common implementation of the DJB hash function, often referred to as DJBX33A. Note the addition operation on line 5.

```
uint djbx33x(char* str, uint len) {
  uint h = 5381;
  uint i = 0;
  for (i=0; i<len; ++i) {
    h=((h<<5)+h)^(str[i]);
  }
  return h;
}
```

Listing 5: A second common implementation of the DJB hash function, often referred to as DJBX33X. Note the exclusive or operation on line 5.

D MULTILINEAR HASH

Lemire introduces a strongly universal set of hash algorithms, named *Multilinear* [32]. For a string $s = s_0 s_1 \dots s_{n-1}$ of length n, the hash value is calculated as

$$h(s) = m_0 + \sum_{i=0}^{n-1} (m_{i+1} s_i)$$

where $m_0, m_1, \dots m_n$ are random values. Every selection of random values defines a different hash algorithm in this set. Since the replacement hash algorithm has to handle strings of any length, using a hash algorithm from the Multilinear set will require us to generate random, or pseudorandom, values on the fly. We can achieve this by incorporating an implementation of a PRNG with the hash function.