FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware

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

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

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

Abstract

We present a static, precise, and scalable technique for finding CVEs (Common Vulnerabilities and Exposures) in stripped firmware images. Our technique is able to efficiently find vulnerabilities in real-world firmware with high accuracy.

Given a vulnerable procedure in an executable binary and a firmware image containing multiple stripped binaries, our goal is to detect possible occurrences of the vulnerable procedure in the firmware image. Due to the variety of architectures and unique tool chains used by vendors, as well as the highly customized nature of firmware, identifying procedures in stripped firmware is extremely challenging.

Vulnerability detection *requires not only pairwise similarity between procedures but also information about the relationships between procedures in the surrounding executable*. This observation serves as the foundation for a novel technique that establishes a partial correspondence between procedures in the two binaries.

We implemented our technique in a tool called `FirmUp` and performed an extensive evaluation over 40 million procedures, over 4 different prevalent architectures, crawled from public vendor firmware images. We discovered *373* vulnerabilities affecting publicly available firmware, *147 of them in the latest available firmware version for the device*. A thorough comparison of `FirmUp` to previous methods shows that it accurately and effectively finds vulnerabilities in firmware, while outperforming the detection rate of the state of the art by 45% on average.

ACM Reference Format:
Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/https://doi.org/10.1145/3173162.3177157

ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/https://doi.org/10.1145/3173162.3177157

1 Introduction

From backbone routers and traffic lights, to home modems and connected refrigerators, embedded devices have an ever-growing presence in modern life.

Embedded devices are closed systems, booting to a consolidated software package known as *firmware*. With the increasing number of IoT devices (expected to reach 20 billion in 2020 [13]) and the constant addition of new features, the total volume of firmware code has grown by orders of magnitude in recent years. Maintaining this wide range of devices and the different versions of firmware images is an extremely challenging task.

Over the years many critical security vulnerabilities were discovered on numerous devices. Notable examples include crippling Distributed Denial of Service (DDoS) attacks on backbone services like Domain Name Service (DNS) [10] and exploits for medical pacemakers [14]. These attacks, and similar potential attacks, have vast implications, including financial damage [11] and even risk to human life.

Moreover, since vendors rely heavily on general-purpose packages integrated in their firmware images, any vulnerability found in these packages may open an entire product line to an attack. This eventuality was throughly explored by Costin et al., [16], who reported a staggering number of 693 vulnerable firmware images by *using meta-data alone*. A responsible company, or a security-savvy individual, should be able to know whether their device is affected by newly published vulnerabilities, yet this is far from being the case.

Firmware is often customized per device Vendors do not provide information about the composition of their firmware, and often the exact composition of the firmware image is not well known even to them [12]. Furthermore, firmware is often customized and optimized to contain only parts of a library to match the particular device on which it runs.

Unpacking the firmware images and exploring the device's file-system usually provides little insight, as the executable files are often stripped from debug information for space considerations. Running the said executables requires command-line access to the architecture and environment of the executable, which is not trivial to attain. Directly searching for the vulnerable assembly code in firmware executables is also challenging, as each vendor may use unique build tool chains, which lead to vast syntactic differences in the assembly.

Problem definition Given a collection of executables $\mathcal{F} = \{T_1, ..., T_n\}$ (e.g., a firmware image), and a query executable Q, containing a (vulnerable) procedure q_v, our goal is to determine for each executable $T_i \in \mathcal{F}$ whether it contains a *similar* procedure to q_v. We define two procedures as similar if they originate from the same source code. Similarity is subject to changes in source code (patching, versions) and declines as procedures differ semantically. Each of the executables in $\mathcal{F}$ may be compiled by any compiler, and can also be stripped of all name information.

Existing Approaches Binary code search is a well-researched problem, but previous approaches are heavily biased towards either end of a spectrum:

- Using several basic blocks or, at best, a single procedure to create a signature for the vulnerability [17–19, 23, 27]. The signature is generated using a combination of the code inside the basic blocks, and the structure of the control flow graph (CFG).
- Comparing whole binaries, attempting to find a full isomorphism for individual CFGs or call-graphs and pinpoint from it the vulnerable procedure [8].

The former approach ignores relevant and valuable information included in the binary, i.e, the surrounding procedures. The latter fails to recognize that modern software is built to conform to different environments and needs, causing wide variance in capabilities (e.g., `wget` can be compiled with or without SSL support, and `curl` can be compiled without cookie support). This leads to vast differences in structure and hinders any chance of reaching full isomorphism. These circumstances cause previous work to suffer from high false positive rates, as we show in our experiments, which include a comparison to (a prominent representative of) each side of the spectrum (`GitZ` [18] and `BinDiff` [8]).

1.1 Our Approach

We propose a new approach to finding similarity over procedures, in the context of the surrounding executable. Our approach is based on the following key ideas:

Representing procedures using canonical fragments We build on our previous work [17] and further improve their *Strands*-based representation. Strands are data-flow slices of basic blocks, canonicalized and normalized to allow for a scalable yet precise cross-{compiler, optimization, architecture} binary code similarity.

We found that the detection of many similar strands is still hindered by syntactic residue, a conclusion we reached by observing the effects of syntactic differences that resulted from vendors using a variety of CPUs and custom build-tools for firmware. To address this, we have further refined the semantics represented by a strand to dissolve such residues. This was achieved by offset elimination and register folding. These adaptations are explained in detail in Section 3.

Using the information in the surrounding executable We broaden the scope of procedure similarity beyond the procedure itself, and observe the neighboring procedures (all other procedures in the executable). This is guided by the observation that procedures always operate within an executable, and will therefore almost always appear with some subset of the neighboring procedures. Using information from neighboring procedures greatly improves accuracy.

Efficiently matching procedures in the context of executables To find procedure similarity within the context of an executable, we require not only to match the procedure $q_v \in Q$ itself to some procedure $t \in T$, but also the surrounding procedures in Q and T (or at least some of them). Producing a full matching over executables containing thousands of procedures is inopportune because finding an optimal full matching is computationally costly.

Instead, we employ an algorithm for producing a partial matching, focused on the query procedure q_v (i.e., the matching need not match all procedures, but must contain q_v). Our algorithm is inspired by model theory's *back-and-forth games* [21], which provide a more efficient replacement for full-matching algorithms when the sets being matched are very large (but not infinite). In our scenario, as matching procedures is a costly operation, performing a full match will not scale to whole datasets of firmware, making a precise partial match a much better approach. We illustrate our game algorithm in Section 2.2 and provide a full description in Section 4.

Main contributions In this paper we make the following contributions:

- We present a novel technique for binary code search, balancing focus between the query procedure and the executable it resides in. The approach is based on robust underlying semantic procedure similarity, and makes use of our key observation that *procedures operate in the context of their executable.*
- We propose an algorithm for using surrounding procedures in the executable to find a more precise matching for a query procedure, through the use of a *back-and-forth game.*
- We implement our approach in a tool called `FirmUp`, built to allow similarity search for executables found in common embedded devices: `MIPS32`, `ARM32`, `PPC32`, and `Intel-x86`. We provide an extensive evaluation of the precision of `FirmUp`, including a detailed comparison to previous work, showing that it achieves substantially better results.
- We used `FirmUp` in our motivating scenario, searching for common vulnerabilities and exposures (CVEs) in over 40 million procedures from firmware images crawled in-the-wild. `FirmUp` was able to correctly locate *373 previously undiscovered vulnerable procedures*

in publicly available firmware, 147 of which *affect up-to-date versions* of the firmware.

2 Overview

In this section, we illustrate our approach informally using an example. Given a query procedure and containing executable, our goal is to find a similar procedure in other stripped executables.

2.1 Pairwise Procedure Similarity

```
1  jalr    t9
2  move    s2, a0
3  move    s5, v0
4  li      v0, 0x1F
5  lw      gp, 0x28+sp
6  bne     s5, v0, 0x40E744
7  move    v0, s5
```

(a) `gcc v.5.2 -O2`

```
1  addiu   a2, sp, 0x20
2  move    s4, a1
3  jal     0x40B2AC
4  move    s5, a0
5  li      v1, 0x1F
6  beq     v0, v1, 0x40B518
7  lui     s6, 0x47
```

(b) `NETGEAR` device firmware

Figure 1. `Wget ftp_retrieve_glob()` vulnerability snippets

The syntactic gap Consider the `MIPS` assembly code snippets in Fig. 1. Although syntactically very different and *sharing no lines of code whatsoever*, the snippets belong to the first basic block (BB) of the same `ftp_glob_retrieve()` procedure from `Wget`. The syntactic difference stems from having compiled the procedures in different settings and with different tool chains. Fig. 1(a) was compiled from `Wget` 1.15, using the `gcc` 5.2 compiler at optimization level 2, while the compilation setting of Fig. 1(b) is *unknown* as it belongs to a stripped firmware image of a `NETGEAR` device, which was crawled from the vendor's public support site. Despite their syntactic variation, the snippets share much of the *same semantics*:

- Both snippets retrieve a value from the stack (line 5 in (a), line 1 in (b)).
- Both load the value `0x1F` and use it in a jump comparison operation (lines 4 and 6 in (a), lines 5 and 6 in (b)).
- Both snippets call a procedure (line 1 in (a), line 3 in (b)).

Although not semantically equivalent, the procedures share great similarity, but finding this similarity is made difficult by the variance in instruction selection, ordering and register usage, as well as different code and data layout (offsets).

Capturing semantic similarity To allow our technique to find similarity over binaries originating from different compilations, we use a procedure representation inspired by [18], and adapt it to our needs. We first decompose a procedure at the BB level (a BB is a node in the procedure's CFG), and further apply slicing [30] at the BB level to generate focused fragments of computation. We then use a compiler optimizer to bring the fragments, which we refer to as *strands*, to succinct canonical form, while also normalizing register names and base address offsets. The representation of a procedure as a set of strands allows capturing of semantic similarity, as the concrete details pertaining to *how* a calculation was performed were abstracted away through the transformation to strands, which reflect only *what* was computed. We define the similarity of a pair of procedures (q, t) as the number of strands shared by the procedures, and denote it $Sim(q, t)$. We further elaborate on the strand creation process in Section 3.

2.2 Efficient Partial Executable Matching

Similarity in the scope of a single procedure is inaccurate

Although procedure-level similarity may achieve fair precision, in some scenarios using the additional information provided by the containing executable can significantly reduce the number of false matches.

Fig. 2 illustrates our approach through the *actual search process* performed for the q_v = `ftp_glob_retrieve()` query found in the Q = `Wget` executable, from Fig. 1(a), searched for in a found-in-the-wild firmware image $\mathcal{F}$, of a `NETGEAR` device, partially featured in Fig. 1(b).

Initially, in Fig. 2(a.0), an attempt is made to match the query procedure with a procedure from a target executable originating from the `NETGEAR` firmware image. Using a naive pairwise similarity based approach, the query procedure will be matched with the procedure in the target executable T with whom it shares the highest similarity score *Sim*, that being the `sub_443ee2()` procedure.

When a broader, executable-level similarity measure is used, as in Fig. 2(a.1-2), this selection is found to be a mismatch, only occurring due to the larger size of `sub_443ee2()`, and because, having undergone different compilations, `ftp_glob_retrieve()` and its true positive `sub_4ea884()`, share fewer strands and thus have a lower *Sim* score. We note that simply normalizing the score by the size of the procedure would not resolve this issue, as the size of a procedure is affected by many factors, most having little to do with the actual semantics. For example, a low optimization level can lead to the creation of a huge amount of code, a size-oriented optimization will shrink it, but a very high optimization can again inflate the code size by loop unrolling. This mismatch demonstrates the limitation of procedure-centric approaches as, for example, `GitZ` [18] would falsely match `ftp_glob_retrieve()` with the `sub_443ee2()` procedure. (We further elaborate in our comparison to `GitZ` in Section 5.3.)

This mismatch can be identified by performing the reverse search, that is, searching for the most similar procedure to `sub_443ee2()` in the query executable (a.1). Doing so results in matching a different procedure– `get_ftp()`, showing that the original match is *inconsistent*, as one would expect to get the same result regardless of the search direction.

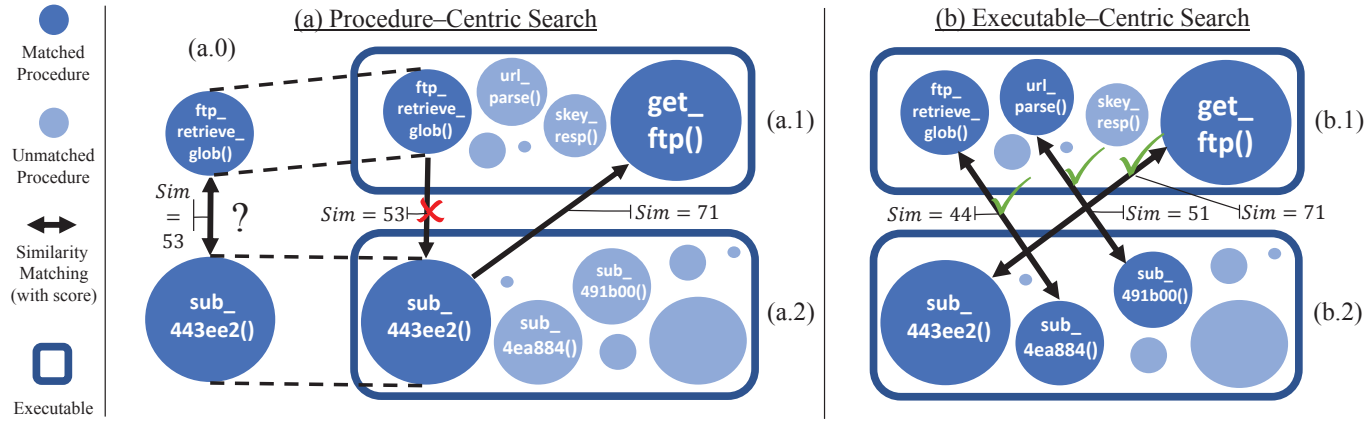


Figure 2. The search process for `Wget`'s vulnerable `ftp_glob_retrieve()` procedure, searched against a found-in-the-wild `NETGEAR` device firmware, in (a) procedure-centric search (on the left) and (b) executable-centric search (on the right)

Another approach to addressing this problem is to match all the procedures and thus establish a full matching between the executables. While a step in the right direction, this approach is severely limited by the assumption that the *whole* structure of the executable is similar, which is not always the case. Major differences in executable structure are often caused by the build configuration selected. For example, in our case the query executable was compiled using default settings, leading to the `skey_resp()`, a procedure handling the OPIE authentication scheme for `sftp`, to be compiled into it, as seen in Fig. 2(a.1). For reasons unknown to us, in this particular instance the vendor compiled `Wget` with the `--disable-opie` option, leading to the omission of this procedure from the target executable. This change can create a "domino effect", causing several procedures to be mismatched. Our approach focuses instead on the query procedure, in an attempt avoid such inconsistent and inaccurate results.

Procedure similarity in the scope of an executable using back-and-forth games Fig. 2(b) illustrates the transition from a procedure-level similarity metric to an executable-level similarity metric, where the scope is broadened by the additional information in the query (b.1) and target (b.2) executables. This result is reached by using an algorithm implementing a back-and-forth game, which establishes and extends a more appropriate *partial* matching, *restricted only by the requirement that it must contain the query procedure.*

Outlining a matching from a two-player game The matching in Fig. 2(b) adheres to the moves of two participants, a *player* and a *rival*, in a back-and-forth game. The game starts by the player picking a matching t_1 = `sub_443ee2()` $\in T$ for the query q_v = `ftp_glob_retrieve()`, passing the turn to the rival, who then tries to pick a better matching for the *target procedure* t_1. The rival selects q_1 = `get_ftp()` $\in Q$ as an alternative and preferable match to t_1, since Sim(`sub_443ee2()`, `get_ftp()`) > Sim(`sub_443ee2()`, `ftp_glob_retrieve()`), forcing the player to reiterate. The game proceeds as described in Tab. 1, until the rival is left with no moves as there are no higher similarity picks for the procedures in the matching. At this point the query procedure q_v is matched with its true positive, t_3 = `sub_4ea884()`. Back-and-forth games, along with an algorithm for producing partial matchings from games, are detailed in Sec. 4.

Actor	Step	Matching
player	Matches q_v = `ftp_glob_retrieve()` with t_1 = `sub_443ee2()`	$\{(q_v, t_1)\}$
rival	Matches t_1 with q_1 = `get_ftp()` $Sim(q_1, t_1) = 71 > 53 = Sim(q_v, t_1)$	~~$\{(q_v, t_1)\}$~~
player	Counters by matching q_1 with t_1 *and* matching q_v with t_2 = `sub_491b00()`	$\{(q_v, t_2), (q_1, t_1)\}$
rival	Matches t_2 with q_2 = `parse_url()` $Sim(q_2, t_2) = 51 > 47 = Sim(q_v, t_2)$	~~$\{(q_v, t_2)$~~, $(q_1, t_1)\}$
player	Counters by matching q_2 with t_2 *and* matching q_v with t_3 = `sub_4ea884()`	$\{(\mathbf{q_v}, \mathbf{t_3}), (\mathbf{q_2}, \mathbf{t_2})\}$ $(\mathbf{q_1}, \mathbf{t_1})\}$
rival	Left with no valid moves. **Game Over**	

Table 1. Game course for Fig. 2

3 Representing Firmware Binaries

3.1 Binary Lifting

From bits to intermediate representation (IR) Stripped executables required the use of mechanisms for lifting bits from various architecture to a more expressive representation, as we need to reason over procedure semantics to find similarity. Using the procedure assembly, which can be extracted (relatively) easily using disassemblers [2, 6], is problematic as assembly instructions are made to be succinct

and not expressive. For instance sub-parts of the same register will appear as differently named variables in the assembly (`mov rax, 0` vs. `mov eax, 0`). Another example is the lack of side-effect expressiveness, for instance in a comparison operation `cmp rax, rbx`, the register flags affected by the operation do not appear in the instruction (or in any of the instructions).

We opted to use the `Angr.io` framework [29] and its `Valgrind` based `VEX` representation [25] as our lifting tool chain. `Angr.io` is a well-maintained and robust framework, containing a wrapper for `Valgrind`'s `VEX-IR`. `Angr.io` offers lifting from various architectures, which enabled us to operate over the most common architectures found throughout our firmware crawling process (Sec. 5.1). The `VEX-IR` contains full representation of the machine state, including side-effects, for each of the translated instructions. We used `IDA Pro` [6] for the parsing and extraction of procedures and BBs from executables, as we noticed that overall it is more accurate when tasked with finding all procedures and blocks in the executable.

Translating embedded architectures Handling four different target architectures, `MIPS32`, `ARM32`, `PPC32` and `Intel-x86`, and specifically executables originating from real firmware images, holds some caveats. First, many of the executables either had a corrupt Executable and Linkable Format (ELF) header, or were distributed with the wrong `ELFCLASS`. Specifically we found that the existence of `MIPS64` executables (8-byte aligned instructions) with a `ELFCLASS32` header is common in firmware. Another caveat, in `MIPS` executables, is the use of a *delay branch slot*, which requires an additional instruction to follow any branch instruction. This additional instruction will be executed while the branch destination is being resolved. This results in the first instruction of the subsequent block being omitted from it and placed as part of the preceding block, which leads to strand discrepancy. Finally, as mentioned, binary lifting tools may still fail to identify several blocks in some procedure, or even omit entire procedures altogether. This is exacerbated in the stripped scenario. We implemented means for overcoming these caveats and further corroborating the results of the lifter, to avoid erroneous results. We added checks for CFG connectivity and coverage of unaccounted-for areas in the `text` section of the ELF file. In an attempt to improve the underlying tools, we provided their creators with our implementations and findings, which were then often adapted into the tool as a patch or a feature.

3.2 Procedure Decomposition

Procedures to strands Our decomposition of procedures is based on CFG representation, where we initially decompose each procedure at BB level. A BB may contain instructions which relate to different executions but reside together due to compiler considerations. Thus we further decompose BBs to independent units of execution by applying slicing [30]. This results in several sub-blocks, each containing only the instructions needed for computing a single output (the Use-Def chain for an outward facing value computed in the BB). This composition is inspired by previous work [17], and thus we also refer to these sub-blocks as *Strands*. Alg. 1 contains the algorithm for the block decomposition.

Algorithm 1: Procedure Decomposition

Input: bb – the BB's instructions (in SSA, as a `List`)
Output: *strands* – the resulting strands

```
1  strands ← ∅, indexes ← {0, 1, ..., |bb| − 1}
2  while |indexes| > 0 do
3      top ← Max(indexes)
4      indexes \ = {top}
5      (s ← List()).append(bb[top])
6      svars ← RSet(bb[top])
7      for i ← (top − 1) .. 0 do
8          if WSet(bb[i]) ∩ svars ≠ ∅ then
9              s.append (bb[i])
10             svars ∪ = RSet(bb[i])
11             indexes \ = {i}
12     strands ∪ = {s}
```

In Alg. 1, each BB is sliced until all values are covered. We assume the BB is in Single Static Assignment (SSA) form, a property of the `VEX-IR` lifting we use. We initialize the process by defining the set of all instructions (*indexes*) in the BB. We iterate over this set, where in every step the last uncovered instruction so far (*top*) is selected. *top* is used as the basis for a new strand (s). The strand is built by iterating backwards over the BB's instructions and adding each instruction only if it defines a variable used by s. In the notation of Alg. 1 we use `RSet`(i) as the set of variables read (or used) by an instruction i, and `WSet`(i) as the set of variables it writes (or defines). In any case, once an instruction is covered by any strand, it is removed (by index) from *indexes*. As we handle each BB separately, the inputs for a block are variables (registers and memory locations) used before they are defined in the block.

3.2.1 Optimizing and Normalizing Strands

To overcome syntactic differences between different compilations of the same procedure, we further operate to bring semantically equivalent strands to the same syntactic form. Capturing similarity through syntactic comparison is crucial to our technique's ability to detect similar strands. We apply the following normalization and optimization operations to bring strands to a succinct, canonical form:

Offset elimination The first step towards canonical form is the removal of offset values that pertain to the concrete structure of the binary file. The removal is performed on the strands produced by Alg. 1. This includes jump addresses

and addresses pointing towards static sections (e.g. `.data`) in the executable. We do not remove offsets which pertain to stack and struct manipulation, as they are more relevant to the semantics of the procedure, serving as a descriptor of the type of data the procedure handles.

Register folding Registers which are read before written to are translated as the arguments to the procedure, and the last value computed by the strand is the return value. We note that even if the return value from a procedure is stored in a register, the value returned must be first generated in the procedure, and so will be captured in the strand.

Compiler optimization Next, we apply the `LLVM opt` optimizer on each of the strands from the previous offset elimination step. This is enabled by a translation of `VEX-IR` strands to `LLVM-IR`. Each strand is translated to a `LLVM-IR` function. The use of the full-blown modern optimizer allows the transformation of strands to a succinct and canonical form, which is useful for finding similarity. Relevant optimizations include expression simplification, constant folding and propagation, instruction combining, common subexpression elimination and dead code elimination.

Variable name normalization To further advance a strand towards canonical form, we rename variables appearing in the optimized strand according to the order in which they appear. This technique is inspired by previous work aimed at finding equivalence over computations [15]. The output of this final stage are the canonical strands we use for our pairwise procedure similarity metric. We denote $Strands(p)$ as the set of canonical strands (each strand is represented as a *string* of its instructions) for a procedure p. Any reference to strands from this point onwards would be to the canonical strands produced by this stage.

Fig. 3 shows an example of a (single) strand extracted from the BB in Fig. 1(a) and the corresponding lifted strand and canonical strand it was transformed to. The strand is responsible for deciding the branch destination. The lifted strand (in the middle) preserves the assembly operation verbatim, and adds a temporary variable for each value operation. Explicit register names and offsets are also kept. After optimizing and normalizing names and offsets (resulting in the bottom strand), the entire operation of the assembly strand (top) is reduced to the first instruction of the `LLVM-IR` canonical strand (bottom), comparing the *normalized* register `reg0` (previously `s5`) with the `0x1F` constant, which was *folded* into the `v0` register. The following instructions reflect the branch operation and either return the *normalized offset* `offset0` or the next program counter value held in `pc`.

3.3 Pairwise Procedure Similarity

After generating the set of strands for each procedure, we compute pairwise similarity. We denote a *query procedure*, i.e., the procedure being searched for, as $q_v \in Q$, where Q is the set of all procedures in the containing *query executable*.

```
move    s5, v0
li      v0, 0x1F
bne     s5, v0, 0x40E744
```

$\downarrow Lift$

```
v0 = external global i32
s5 = external global i32
pc = external global i32
define i32 strand() {
entry:    t0 = v0
          store i32 s5, t0
          t1 = 0x1F
          store i32 v0, t1
          t0 = icmp ne i32 t0, t1
          br i1 t0, label if.true, label if.false
if.true:  ret i32 0x40E744
if.false: ret i32 pc
}
```

$\downarrow Optimize, Normalize$

```
define i32 strand(i32 reg0, i32 offset0, i32 pc) {
entry:    t0 = icmp ne i32 reg0, 0x1F
          br i1 t0, label if.true, label if.false
if.true:  ret i32 offset0
if.false: ret i32 pc
}
```

Figure 3. Assembly strand computing the branch destination result (top) and its lifted `LLVM-IR` strand (middle) and final canonical `LLVM-IR` form (bottom)

The *target procedure*, i.e., the candidate procedure q is being compared to, is denoted by $t \in T$ (similarly, T being the containing *target executable*). Given a pair (q, t), we define procedure similarity as follows:

$$Sim(q, t) = |Strands(q) \cap Strands(t)|,$$

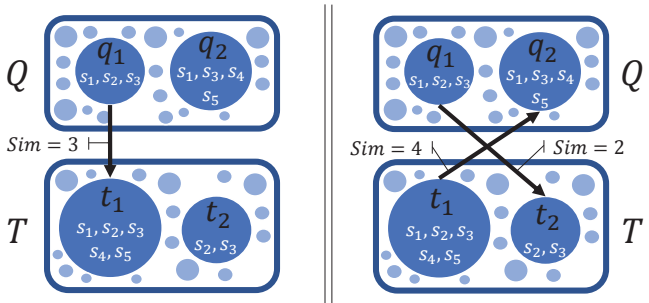
i.e., $Sim(q, t)$ is simply the number of unique canonical strands that are shared in both representations. To calculate Sim faster, we keep the procedure representation as a set of hashed strands (without consideration for hash counts).

4 Binary Similarity as a Back-and-Forth Game

4.1 Game Motivation

Procedure-centric matching is insufficient We first show the intuition for using a game algorithm as a means to find procedure similarity while accounting for the surrounding context. The goal is to transition from a local match, which relies on a local maximum of similarity score, to a global maximum. This is done by composing a matching that is expected to maximize overall similarity across the binaries. Fig. 4 illustrates a conceptual example for matching a procedure $q_1 \in Q$ with a procedure in an executable T. The notation is similar to Fig. 2, where squares represent executables which are sets of procedures and circles are procedures which are sets of strands denoted as s_i.

The procedure-centric approach in Fig. 4(a) immediately matches q_1 with t_1 as it has the most shared strands. We



(a) procedure-centric matching (b) executable-centric matching

Figure 4. Two approaches for finding a similar procedure to $q_1 \in Q$, within the executable T. The procedure-centric approach on the left leads to a less accurate matching. Strands $\{s_1, s_2, s_3, s_4, s_5\}$ are spread over all procedures.

denote $q \underset{Q,T}{\overset{PC}{\sim}} t$ as matching the procedure q with a procedure t that has the maximal similarity ($Sim(q,t)$) score in T. The $q_1 \underset{Q,T}{\overset{PC}{\sim}} t_1$ matching ignores the neighboring procedures in the executables, resulting in a sub-optimal match. The existence of q_2 and t_2, and the higher similarity score of q_2 and t_2, indicate that t_2 is a more correct matching for q_1, in a global view. (This is also the case in Fig. 2(a), resulting in a false matching for a real world example.)

Knowledge in surrounding executable leads to better matching To induce a better matching, we constrain the matching process with the rules of a two-player game, inspired by model theory's notion of back-and-forth games, also called Ehrenfeucht- Fraïssè games [21]. The main premise of the game is that *the player trying to match a procedure q_v is countered with a rival.*

Applying these notions to the example from Fig. 4 creates the following game scenario:

- The player performs its first selection, $t_1 \in T$, as it is the best match for q_1. The rival tries "to show the player it was wrong" by selecting a different procedure $q_2 \in Q$ with a higher similarity score $Sim(q_2, t_1) > Sim(q_1, t_1)$.
- The player corrects its selection by adapting (q_2, t_1) as part of the matching and choosing another procedure $t_2 \in T$ as a pairwise matching for q_1.
- Following this selection, the rival cannot find a better match for t_2 and "forfeits the game".

Formalizing the intuition behind the game

Using $Sim(q,t) = \zeta$, we define $q \underset{Q,T}{\sim} t$ in Eq. 1

$q \underset{Q,T}{\sim} t$ expresses the back-and-forth selection process. In this process the best match for q *will be ignored* if there exist a better match between other procedures in Q and T.

In this definition, each matched procedure pair (q,t) is the best match among all of the procedures in T and Q, while

$$
\begin{gathered}
q \underset{Q,T}{\sim} t \\
\Updownarrow \\
\forall q_i \in Q, q_i \neq q \Rightarrow \\
\left(\left[\begin{array}{l} Sim(q_i,t) < \zeta \ \vee \\ \exists t_{i'} \in T, t_{i'} \neq t, Sim(q_i,t_{i'}) \geq Sim(q_i,t) \Rightarrow q_i \underset{Q,T}{\sim} t_{i'} \end{array} \right] \right) \\
\wedge \\
\forall t_j \in T, t_j \neq t \Rightarrow \\
\left(\left[\begin{array}{l} Sim(q,t_j) < \zeta \ \vee \\ \exists q_{j'} \in Q, q_{j'} \neq q, Sim(q_{j'},t_j) \geq Sim(q,t_j) \Rightarrow q_{j'} \underset{Q,T}{\sim} t_j \end{array} \right] \right)
\end{gathered}
\tag{1}
$$

other procedures are explicitly allowed to be a better match *if and only if* these other procedures have a different better match themselves. The first big parentheses express the option that another procedure, $q_i \neq q$, is a better *local match* for t, meaning that $Sim(q,t) < Sim(q_i,t)$. This is allowed as long as q_i has another better *global match* $t_{i'} \in T$, so that $q_i \underset{Q,T}{\sim} t_{i'}$. The second big parentheses express the same possibility for the other direction. There can be $t_j \neq t$ for which $Sim(q,t) < Sim(q,t_j)$, as long as t_j has another better match $q_{j'} \in Q$, so that $q_{j'} \underset{Q,T}{\sim} t_j$.

It is important to note that these requirements do not force a full matching between the executables; they will only create the needed additional matches between the executable procedures, to allow for a consistent match (and for the player to win).

Implementing the matching algorithm according to the aforementioned game, where the player aims to win and thus needs to account for the context of the whole executable, leads to better matchings.

4.2 Game Algorithm

Alg. 2 details our similarity game algorithm. Because the algorithm implements *the winning strategy* for the *player* trying to match a query, this is not an explicit two-step algorithm performed by the two players.

The algorithm accepts as input the query executable, query procedure, and target executable. Each executable is represented as a set of procedures, and each procedure is represented by the set of its strands extracted as described in Section 3.

In line 1 we create an empty dictionary, *Matches*, which will accumulate all the matched procedures, i.e., procedures $q \in Q, t \in T$ for which $q \underset{Q,T}{\sim} t$. In line 2 we initialize a stack, *ToMatch*, to hold all the procedures we are trying to match. In this process we also push q_v as this is the primary procedure we need to match.

Following the initializations we start the matching game, expressed as a while loop, which ends when the game ends.

Algorithm 2: Similarity Game Algorithm

Input: T – The target executable procedures
Q – The query executable containing q_v
q_v – The vulnerable procedure in Q
Output: *Matches* – The resulting partial matching, containing (at least) a matching for q_v

```
1  Matches ← {} , ToMatch ← Stack()
2  ToMatch.Push(q_v)
3  while GameDidntEnd() do
4      M ← ToMatch.Peek()
5      if M ∈ Q then
6          My,Other ← Q,T
7      else
8          My,Other ← T,Q
9      Forward ← GetBestMatch(M,Other,Matches)
10     Back ← GetBestMatch(Forward,My,Matches)
11     if M== Back then
12         Matches+= M ↔ Forward
13         ToMatch.Pop (Matches)
14     else
15         ToMatch.PushIfNotExists([Forward,Back])
```

Game ending conditions are checked by the `GameDidntEnd()` procedure, to be described later. In each loop iteration, or step in the game, we try to match the procedure at the head of the *ToMatch* stack (retrieved by `Peek()` in line 4). We resolve in which direction we will perform the match, by checking whether it is part of Q or T, and setting *My* and *Other*. We perform a *forward match*, i.e., we search for the best match for M in *Other*, while ignoring all previously matched procedures. This is done by calling `GetBestMatch()` (line 9). Using the best match for M, *Forward*, we perform the *backwards match*, and store the result in *Back* (line 10). Collecting the best matches in both directions gives us two possibilities:

- *Back* is M, meaning that $M \underset{Q,T}{\sim} Forward$ (checked in line 11). In that case this match is added to *Matches*, and M is popped from the stack (lines 12-13).
- Otherwise, we need to find a match for *Forward or Back* before we can claim that we have a match for M. To do that, in `PushIfNotExists()`, we check if *Forward* and *Back* are already in the stack; otherwise we push the missing procedures.

The implementation of `GameDidntEnd()` will check for one of the following conditions:

- A match was found for q_v
- *ToMatch* has arrived at a fixed state. This will happen when no new procedures are pushed to *ToMatch* at `PushIfNotExists()` (line 15). In this case the game will never end, meaning that the matching process will not succeed.
- As a heuristic, the game can also be stopped if too many matches were found or *ToMatch* contains too many procedures.

4.3 Graph-based Approaches

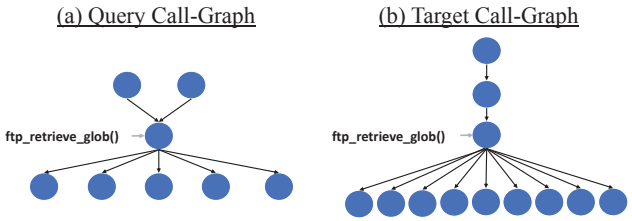


Figure 5. The call-graphs surrounding `Wget`'s `ftp_retrieve_glob()` procedure in the `MIPS` query (left) and a target from a `NETGEAR` firmware (right). The variance in call-graph structure prevents graph based techniques (e.g., `BinDiff`) from succeeding.

Limitations of the graph-based approaches Using a graph-based technique, which tries to leverage relationships over the procedures in the form of a call-graph, is problematic in our scenario. Fig. 5 depicts two samples of call-graphs, restricted to one level below and two levels above the `ftp_retrieve_glob()` procedure from `Wget`. The sample on the left belongs to the `MIPS` query executable, and the one on the right originated from a `NETGEAR` firmware. Even in this limited scope, the variance in call-graph structure is vast. This can be attributed to the customized nature of firmware, along with compiler inlining and dynamic call targets, which produce high variance over the target call-graph, reducing the accuracy of a graph-based matching. This is further demonstrated in our evaluation (Section 5.3), showing that `BinDiff`'s graph-based technique results in poor precision.

5 Evaluation

We evaluate our approach using `FirmUp` in the scenario of identifying vulnerable procedures found in real-world firmware images, and comparing it to prominent existing work in the same scenario. The evaluation is thus composed of (i) a description of the queries and corpus creation process (Sec. 5.1), (ii) our main experiment, in which vulnerable procedures were located in publicly available firmware images (Sec. 5.2), and (iii) a controlled experiment evaluating `FirmUp`'s accuracy alone and in comparison to `BinDiff` and `GitZ` (Sec. 5.3).

5.1 Corpus Creation

Into the wild To simulate a real-world vulnerability search scenario, we created a crawler to collect, unpack, organize

and index executables in firmware. Our crawler downloads firmware images from publicly available repositories of prominent device vendors, including NETGEAR [7], D-Link [4] and ASUS [1]. We used binwalk [5] for unpacking firmware images. Out of ~5000 firmware images collected, only ~2000 produced usable executables, while others failed to unpack or consisted only of content or configuration.

After unpacking, the crawled images amounted to ~200,000 executables. Each executable contained ~200 procedures on average, resulting in a dataset containing ~40,000,000 procedures. Finally, the procedures were indexed as a set of strands (as described in Section 3.2).

Experimental setup Experiments were performed on an Ubuntu 16.4 machine with two Xeon E5-2699V3 CPUs (36 cores, 72 threads), 368 GiB of RAM, 14 TB HDD. The overall memory consumption for each thread did not exceed 560 MiB.

Using prominent open-source CVEs as queries To evaluate FirmUp's ability to locate vulnerable procedures, we gathered query procedures known to be vulnerable to some attack according to a CVE report. We put an emphasis on widely distributed software as: (i) we wanted to adhere to a real-world scenario and (ii) this made the procedure more likely to be used by some device and thus be found in our corpus of crawled firmware. To produce our search query, we used the latest vulnerable version of the software package as reported by the CVE and compiled it with gcc 5.2 at the default optimization level (usually -O2). We intentionally chose a range of diverse procedures, affected by different vulnerability types, including DoS due to crafted message (Tab. 2 lines 1,5), BOF (Tab. 2 lines 2,7), input validation (Tab. 2 line 3), information disclosure (Tab. 2 line 4), and path traversal (Tab. 2 line 6).

5.2 Finding Undiscovered CVEs in Firmware

Tab. 2 shows the results of our experiments with found-in-the-wild firmware images. *373* vulnerable procedures were found in publicly available firmware images, *147 of them in the latest available firmware version for the device*. We note that these experiments included stripped procedures only, to adhere to a real-world scenario. The use of stripped firmware also led to new findings, pointing to the exact location of the vulnerable procedure.

Each line in Tab. 2 depicts an experiment in which a vulnerable procedure was searched for in our corpus of firmware. Line 1 of the table, for instance, depicts the results of the experiment designed to search for the vulnerable procedure in CVE-2011-0762. This vulnerability was discovered in the "Very Secure FTP Daemon" (vsftpd) package. The relevant executable from the package, the FTP server (aptly named vsftpd), was found by FirmUp in 76 firmware images. In this experiment our engine correctly found the vulnerable vsf_filename_passes_filter() procedure in *all* of the suspect executables. The false positives (FPs) column measures the number of procedures wrongly matched by FirmUp to the vulnerable query, which is 0 for this experiment. The next column lists the vendors for the devices in which the CVE was discovered, and the column to the right gives the number of devices for which the *latest firmware version available is vulnerable*. The rightmost column shows average overall (user+kernel mode) run time, across three runs, for that experiment.

Confirming findings Procedures matched by FirmUp to a vulnerable query procedure were confirmed in a semi-manual fashion. We used a combination of markers such as string constants, use of global memory, structures access, and others more specific to the vulnerable procedure. Additional matched procedures (excluding the query) were used to further assist the validation process. We then grouped procedures according to their effective address in the executable (the exact same executable sometimes appeared in several images) and manually verified the result. We found that subsequent versions of firmware images targeting the same device will sometimes not re-compile executables if they are not part of the software updated in that release.

We note that immense manual effort would be required to rule out vulnerable procedures not found by FirmUp in this wild objects scenario. Instead, to measure the precision of our tool, we performed a controlled experiment described in Sec. 5.3.

Noteworthy findings *False positives*: Line 6 of Tab. 2 shows the experiment with the highest false positive rate overall (and the only experiment to report false positives). We used the wget version 1.15 executable as the query sample. Analysis of the results showed that the 14 FPs were a result of discrepancies between versions, e.g., the target executable was a previous (1.12) version of wget. The notable semantic (and subsequently syntactic) differences over the versions of the tool were the main reason that the query procedure was falsely matched with an unrelated target procedure.
Deprecated procedures: Line 3 of Tab. 2 details the result for the curl_easy_unescape() procedure in libcurl. FirmUp claimed to locate one supposedly non-stripped sample of this procedure. This was an unexpected result, as curl_easy_unescape() is *exported* in libcurl. In depth examination uncovered that this is not a stripping error but an example of a vendor using an old version of curl containing a predecessor of the query procedure– curl_unescape(), which was long since deprecated. The deprecated procedure in this experiment matched our query. We were surprised to learn that vendors sometimes use very outdated versions of software in their firmware, despite many CVEs having been issued for them (more than 39! [3]). The firmware in question was released in March 2014; however, the curl_unescape() procedure was deprecated in *June 2006* (version 15.4).

#	CVE	Package	Procedure	**Confirmed**	FPs	Affected Vendors	**Latest**	Time
1	2011-0762	vsftpd	vsf_filename_passes_filter	**76**	0	ASUS,NETGEAR	**29**	2m
2	2009-4593	bftpd	bftpdutmp_log	**63**	0	NETGEAR	**15**	4m
3	2012-0036	libcurl	curl_easy_unescape	**1**	0	NETGEAR	**0**	12s
4	2013-1944	libcurl	tailmatch	**5**	0	ASUS,D-Link	**2**	1m
5	2013-2168	dbus	printf_string_upper_bound	**10**	0	D-Link,NETGEAR	**5**	7m
6	2014-4877	wget	ftp_retrieve_glob	**69**	14	ASUS,NETGEAR	**35**	18m
7	2016-8618	libcurl	alloc_addbyter	**149**	0	ASUS,D-Link,NETGEAR	**61**	25m

Table 2. Confirmed vulnerable procedures found by FirmUp in publicly available, stripped firmware images

5.3 Evaluating Precision Against Previous Work

Refining Wild Data for a Controlled Experiment

We designed a controlled experiment to accurately evaluate our approach and enable a fair comparison to previous work. We decided to avoid the use of synthesized targets, as it was infeasible to create a collection of tool chains diversified enough to accurately approximate the wild data. Instead, we used a subset of a corpus containing executables where the vulnerable procedures were labeled. These were split into two groups:

1. Non-stripped executables - these were not very common, as most firmware software is stripped to save space. Some non-stripped versions were found in early releases, probably left there to allow for debugging.
2. Exported procedures - when the query is an exported procedure, it can be easily located in the binary even when the executable is stripped. This case is very common for libraries where many procedures are exported, for example the procedure exif_entry_get_value() from libexif, which was affected by CVE-2012-2841 but does not appear in Tab. 2 as the table includes results over stripped procedures only.

Limiting our experiment to these types of procedures reduced our overall number of targets, but allowed us to accurately report precision for each tool. To increase the scope of the experiment, we added two more CVEs belonging to the second group from the libexif and net-snmp code packages.

***Comparison to* BinDiff** First we experimented with BinDiff, which is the de facto industry standard [8]. BinDiff aims to find similarity between procedures in whole binaries. It operates over two binaries at a time (a query and a target), scanning the procedures of both input files and producing a mapping of the two binaries. This mapping matches a subset of the procedures in one binary with a subset of the other, using the control structure of the procedures, call graph and syntactic matching of instructions (further information can be found in [9]).

Note that for this scenario we could only run a subset of our experiment, from the first group of labeled data described above (non-stripped, non-exported). The other group could not be used as BinDiff, in its similarity score calculation process, attributes great importance to the procedure name when it exists. As we found no visible way of configuring BinDiff to ignore this information, we could not objectively assess its accuracy for these stripped scenarios.

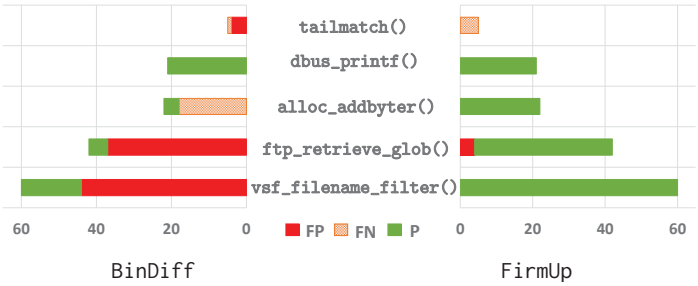
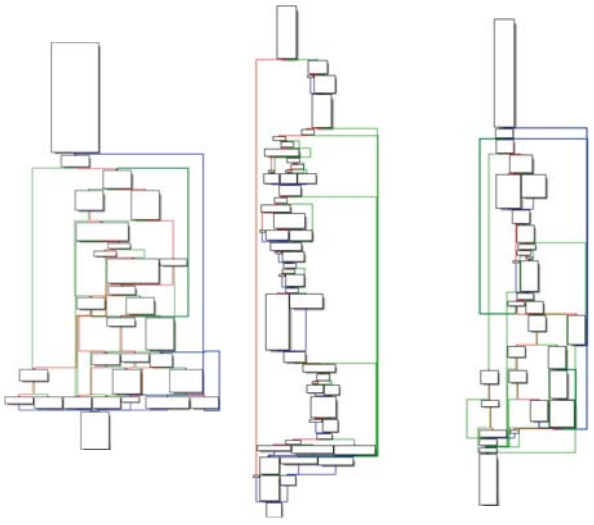


Figure 6. Labeled experiment comparing BinDiff with FirmUp, showing that BinDiff encountered over 69.3% false results overall compared with 6% for FirmUp

Fig. 6 shows the results of the labeled experiments, using FirmUp (a) and BinDiff (b). Each line represents an aggregation of all of the results reported by each tool for a given query against all the relevant targets, split to (true) positive, false negative and false positive, marked respectively as (P), (FN) and (FP). Note that for BinDiff we consider an unmatched procedure to be a false positive (because we know it is there), and a matched procedure to be positive or false positive depending on the correctness of the match.

Aggregating the results, we see that BinDiff suffers from a very high FP rate, mismatching 85 out of 150 targets (FPr = 56.6%). It further suffers from a false negative rate of 12%, and only 32% positive matches.

For the same dataset, FirmUp successfully matches 141 targets, achieving a 96% success rate, with only 4 (2.6%) false positives and 5 (3.33%) false negatives. The biggest difference in accuracy was measured in the vsf_filename_filter() experiment (bottom line). BinDiff succeeded in matching 26% of the targets, yet still trailed behind FirmUp, which achieved a perfect result. In the next experiment, ftp_retrieve_glob() (one line above), BinDiff detects 5 procedures, yet falsely matches all of the rest, 37 out of 42. In this case FirmUp successfully identifies 38 targets.



(a) FirmUp: t_v ✓ **(b)** q_v **(c)** BinDiff: t_j ✗

Figure 7. BinDiff's disproportionate focus on the procedure's internal structure leads to false matches for `vsf_filename_filter()`

To better understand why BinDiff performed so poorly in the `vsf_filename_filter()` experiment (the bottom line in Fig. 6), we more closely examined the one false positive it reported. Fig. 7(b) shows the CFG of the vulnerable procedure, q_v. In this graph the nodes are the basic blocks, shown without their contents (the code) for readability. The edges are colored green and red for true and false branches respectively, and blue for unconditional control-flow. The CFG of Fig. 7(c) closely resembles that of Fig. 7(b), causing BinDiff to report it as a match. This is a false match as this CFG represents an unrelated procedure in T which we denote by t_j. On the other hand, FirmUp performs a semantic comparison, relying on shared canonical strands in the blocks of q_v and t_v and correctly reports Fig. 7 (a) as the best match.

***Comparison to* GitZ** Next, we experimented with GitZ [18]. We note that GitZ was not designed for our testing scenario, as it compares procedures while disregarding the origin executable. Moreover, there is no notion of a positive or negative match; instead, GitZ accepts a single query and a set of targets and returns an ordered list of decreasingly similar procedures from the targets.

In the comparison experiment, we used each query against all the procedures in each target executable, and considered the first result (top-1) to conclude whether the match was a positive or a negative. This result is presented in Fig. 8. We further explored accuracy for the top-k results produced by GitZ; however, we note that this scenario is much more time consuming for a human researcher. Thus we did not include the top-k results of FirmUp in any of the results. The top-k comparison is reflected in Fig. 9, as explained below.

Figure 8. Labeled experiment comparing GitZ with FirmUp, showing that GitZ encountered 34% false positives overall

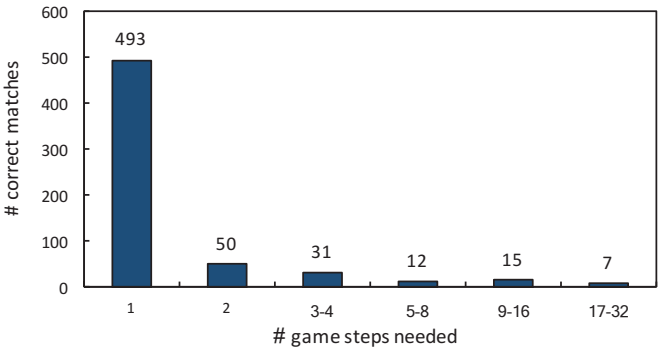


Figure 9. Number of procedures correctly matched as a factor of the number of steps in the back-and-forth game. 180 procedures required more than 1 step for the matching, up to 32 steps.

To maximize GitZ's precision and allow for a fair comparison, we trained a global context (a set of randomly sampled of procedures in the wild used to statistically estimate the significance of a strand; see [18] for more details) for each architecture separately, using more than a thousand procedures for each one, and using the right context for each comparison. Note that for this experiment we used queries from both groups mentioned above.

Fig. 8 shows the results of the labeled experiments for GitZ (a) and FirmUp (b), similarly to Fig. 6. Note that in this comparison we treated false positives and false negatives together, marking them as false positives in the figure. GitZ suffers from relatively poor precision, reporting 274 false positives, compared with 78 false positives and false negatives combined reported by FirmUp, and shown in the figure as false positives. For example, in the `ftp_retrieve_glob` experiment in line 4 GitZ does not report any true positives, while FirmUp successfully detects 38 of the 42 targets. Furthermore, in the `curl_easy_unescape()` experiment in line 8, GitZ only managed to detect 104 targets out of 264, while FirmUp detected 171.

Next, we examine the contribution of our back-and-forth approach.

Fig. 9 shows number of game steps required to perform matches. While 493 of the queries could be matched in one

iteration of the game algorithm, 115 procedures (or 19% of the 608 targets matched in one iteration of the game algorithm) required an additional $1-32$ *more steps* to arrive at the positive match. When a single iteration was sufficient, strands were shown to be effective in bridging the syntactic gap over the procedures. When more steps were needed, a matching process (as in Tab. 1) took place to account for surrounding procedures, correcting matches accordingly. These cases stem from a range of scenarios, including:

- Very large procedures that are mistakenly matched with the query due to their size.
- The number of shared strands between the query and the true positive was very small.
- The true positive and several false positives share an equal number of strands with the query.

These cases are resolved in our algorithm, which effectively searches for replacement matches for all candidates, thus eliminating wrong matches. Without this iterative matching process, the overall precision drops from 90.11% to 67.3%.

Note that Fig. 9 allows us to evaluate `FirmUp` vs. `GitZ` while considering the top-k results. Since both tools rely on shared strands (with further optimizations performed in `FirmUp` as described in Sec. 3.2), the top-k accuracy of `GitZ` can be approximated by aggregating the columns of Fig. 9. For instance by observing the 2 game-steps column in Fig. 9, we can see that considering the top-2 results from `GitZ` will reduce the number of false positives by approximately 50.

6 Related Work

Vulnerability search in firmware The authors of [16] were the first to direct attention to security of embedded firmware, in the context of locating CVEs. The main goal of this work was to obtain a quantitative result, as they did not perform any sophisticated static analysis. Firmware security was further explored in [24], which used statistical and structural features of the procedure in order to locate vulnerabilities in firmware. Their heuristic-based, procedure-centric approach performs a rather shallow analysis, which is demonstrated by the low detection rate (38 findings for a corpus of 420 million procedures and 154 vulnerabilities).

Procedure-centric static approaches Tracelet-based code search [19] is a syntactic based approach for procedure similarity, with the addition of a rewrite engine. This approach was shown to be less accurate than semantic approaches [18]. However, the use of a highly semantic biased approach is in itself problematic, resulting infeasible run-time, making the approach inapplicable to the large scale firmware search scenario. `Expose` [26] attempts to apply a balanced approach over semantics and syntax, using symbolic execution as well as syntactic heuristics. While this compromise shows potential, accuracy is poor, although this can also be attributed to the procedure-centric nature of the approach.

The authors of [28] use expression trees as procedure signatures, but this approach is susceptible to the problem of varying compilation tool-chains as different compilers perform the same calculation differently.

The authors of [27] present an approach which samples `VEX-IR` lifted and simplified (using theorem-provers) basic-blocks. To avoid over-matching, this work uses a sampling-based approach to accompany the simplified expressions. This approach suffers from lower precision and recall than FirmUp. We attribute this to the spurious matches that result from common computations shared among non-similar code. These events were observed in our experiments, leading us to employ a statistical framework instead. Also note that our approach is purely static.

The authors of [22] leverage the (static) CFG of the procedure for similarity, while applying a dynamic filtering beforehand to scale. Their results show that the CFG in not a precise enough feature for similarity, leading to false positives.

Dynamic approaches Blanket execution [20] is a technique for producing procedure signatures by collecting run time side-effects of the procedure on the environment. This results in sub-optimal accuracy; applying a dynamic approach is moreover problematic in the embedded scenario because acquiring a proper running environment is nontrivial, and may expose the analyzing party to unnecessary risk.

7 Conclusion

We present a static precise and scalable technique for finding CVEs in stripped firmware images. When dealing with CVEs in firmware, the common approach of creating a procedure-centric signature (e.g., [17–19, 23, 27]) yields poor precision because it fails to leverage other information present in the binary. The other commonly used approach of comparing full binaries ([8]) also falls short due to the customization of firmware images and call-graph variance.

Our middle ground technique establishes correspondence between *sets of procedures* in a given query binary and target binary. We show that the pairwise similarity of procedures cannot be lifted directly to establish this correspondence. We therefore introduce a back-and-forth game to lift the pairwise similarity between procedures to similarity between sets of procedures.

We implemented our technique and extensively evaluated it over millions of procedures from public vendor firmware images. We discovered 373 vulnerabilities in publicly available firmware, 147 of which appear in the latest available firmware version for the device. We further compared our approach to previous methods and show that we are able to accurately find vulnerabilities in firmware, while outperforming the detection rate of the state of the art by 45% on average.

Acknowledgments

We would like to thank Santosh Pande, for helping us improve and clarify the paper throughout the shepherding process. Our thanks also goes out to Omer Katz for his through feedback for this work, as well as the anonymous referees for their comments and suggestions. The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC-COG-PRIME and the Israel Ministry of Science and Technology, grant no. 3-9779.

References

[1] [n. d.]. ASUS Firmware Resources. https://www.asus.com/support/.
[2] [n. d.]. Capstone Disassembler. www.capstone-engine.org/.
[3] [n. d.]. curl releases. https://curl.haxx.se/docs/releases.html.
[4] [n. d.]. D-Link Firmware Resources. http://support.dlink.com/.
[5] [n. d.]. Firmware Analysis. https://github.com/devttys0/binwalk
[6] [n. d.]. Hex-Rays IDAPRO. http://www.hex-rays.com.
[7] [n. d.]. Netgear Firmware Resources. downloadcenter.NETGEAR.com/.
[8] [n. d.]. zynamics BinDiff. http://www.zynamics.com/bindiff.html.
[9] [n. d.]. zynamics BinDiff Manual. http://www.zynamics.com/bindiff/manual/index.html.
[10] 2016. Dyn cyberattack. https://en.wikipedia.org/wiki/2016_Dyn_cyberattack.
[11] 2016. Muddy Waters Hedge Fund Claims Device Maker Vulnerable to Hackers. https://www.ft.com/content/bfde006a-6b0d-11e6-ae5b-a7cc5dd5a28c?mhq5j=e3.
[12] 2016. Updates and more on the Netgear router vulnerability. http://www.computerworld.com/article/3151097.
[13] 2017. Gartner: 8.4 Billion Connected "Things" Will Be in Use in 2017. http://www.gartner.com/newsroom/id/3598917.
[14] 2017. Over 8,600 Vulnerabilities Found in Pacemakers - The Hacker News. (2017). http://thehackernews.com/2017/06/pacemaker-vulnerability.html.
[15] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. 1997. Value Numbering. *Software: Practice and Experience* 27, 6 (June 1997), 701–724. https://doi.org/10.1002/(SICI)1097-024X(199706)27:6<701::AID-SPE104>3.3.CO;2-S
[16] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A Large-scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 95–110.
[17] Y. David, N. Partush, and E. Yahav. 2016. Statistical similarity of binaries. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vol. 13-17-June. https://doi.org/10.1145/2908080.2908126
[18] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*. ACM Press, New York, New York, USA, 79–94. https://doi.org/10.1145/3062341.3062387
[19] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 349–360. https://doi.org/10.1145/2594291.2594343
[20] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22*. 303–317. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele
[21] Andrzej Ehrenfeucht. 1961. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae* 49, 2 (1961), 129–141. http://eudml.org/doc/213582
[22] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.
[23] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 480–491. https://doi.org/10.1145/2976749.2978370
[24] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 480–491. https://doi.org/10.1145/2976749.2978370
[25] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*. 89–100.
[26] Beng Heng Ng and A. Prakash. 2013. Expose: Discovering Potential Binary Code Re-use. In *IEEE 37th Annual, Computer Software and Applications Conference (COMPSAC), 2013*. 492–501. https://doi.org/10.1109/COMPSAC.2013.83
[27] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 709–724. https://doi.org/10.1109/SP.2015.49
[28] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New York, NY, USA, 406–415.
[29] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2016).
[30] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (jul 1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248