Extracting Conditional Formulas for Cross-Platform Bug Search

Qian Feng[†], Minghua Wang [◊], Mu Zhang[‡], Rundong Zhou[†], Andrew Henderson[†], and Heng Yin[*]

[†]Department of EECS, Syracuse University, USA
[◊]BaiDu X-Lab
[‡]NEC Laboratories America
[*]University of California, Riverside
[†]{qifeng,rzhou02,anhender}@syr.edu
[‡]mu@nec-labs.com [◊]wangminghua01@baidu.com [*]heng@cs.ucr.edu

ABSTRACT

With the recent increase in security breaches in embedded systems and IoT devices, it becomes increasingly important to search for vulnerabilities directly in binary executables in a cross-platform setting. However, very little has been explored in this domain. The existing efforts are prone to producing considerable false positives, and their results cannot provide explainable evidence for human analysts to eliminate these false positives. In this paper, we propose to extract conditional formulas as higher-level semantic features from the raw binary code to conduct the code search. A conditional formula explicitly captures two cardinal factors of a bug: 1) erroneous data dependencies and 2) missing or invalid condition checks. As a result, binary code search on conditional formulas produces significantly higher accuracy and provide meaningful evidence for human analysts to further examine the search results. We have implemented a prototype, XMATCH, and evaluated it using well-known software, including `OpenSSL` and `BusyBox`. Experimental results have shown that XMATCH outperforms the existing bug search techniques in terms of accuracy. Moreover, by evaluating 5 recent vulnerabilities, XMATCH provides clear evidence for human analysts to determine if a matched candidate is indeed vulnerable or has been patched.

Keywords

Vulnerability Search, Binary Analysis, Firmware Security

1. INTRODUCTION

Recent security breaches in embedded systems and IoT devices lead to a growing focus on vulnerability detection directly in binary executables across multiple platforms. That is, when a vulnerable function is identified in a piece of binary code for a particular platform (e.g. x86), the interest is in determining whether a binary executable for a different platform (e.g. ARM or MIPS) contains the same vulnerable function. Generally, this belongs to fundamental research area called cross-platform binary code search.

Existing efforts have been made to demonstrate the feasibility of cross-platform binary code search. By exploring the intrinsic features shared by different architectures, existing methods are able to achieve a reasonable accuracy [19,35]. For example, discovRe [19] utilizes a set of learned syntactic features as filtering to guide the search to refine the similarity matching results. The work by Pewny et al. [35] extracts the semantic feature, input and output values for each basic block to improve the accuracy of CFG matching.

However, the existing techniques tend to produce a considerable number of false positives. Therefore, even if they could assign high ranks for the correct target functions based on similarity scores, a security analyst would have to manually examine each candidate. This manual process is expensive, tedious, and error-prone.

To further improve search accuracy and facilitate subsequent manual examination, we argue that higher-level semantic features must be recovered from the raw binary code. Ideally, if we could *completely* recover the original source code and perform source-code level search, we could achieve 100% accuracy and provide rich information for human analysts to examine the searched results. Unfortunately, this ideal solution is impractical, because decompilation is still far from being mature.

In this paper, we introduce a novel semantic feature called "conditional formula", as a middle ground between binary-level syntactic features and source-code level representation. To conduct code search, we factorize tangled code logic of a vulnerable function down into conditional formulas as logic-independent units. Generally, a conditional formula consists of an If-clause and a Then-clause, and each clause is a symbolic formula, describing under what condition (stated in the If-clause) a given action (in the Then-clause) will take place. A conditional formula explicitly captures two cardinal factors of a buggy code: (1) erroneous data dependencies, and (2) missing or incorrect condition checks. Instead of treating the vulnerable function as a whole, searching on structured conditional formulas can effectively localize the possibly vulnerable code logic. By contrasting conditional formulas between the vulnerable function and a target candidate, an analyst can quickly diagnose whether the target is vulnerable or a false positive.

Therefore, using conditional formulas for code search has two advantages: (1) it significantly improves search accuracy, as it abstracts away platform-specific differences at the binary code level; and (2) it provides explainable evidence for human analysts to scrutinize the search results and identify vulnerable functions.

More specifically, to extract conditional formulas, we lift the binary code into a platform-independent intermediate representation (IR) and conduct static analysis on the IR. We formulate the matching of conditional formulas as a linear assignment problem

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates
© 2017 ACM. ISBN 978-1-4503-4944-4/17/04. . . $15.00
DOI: http://dx.doi.org/10.1145/3052973.3052995

and leverage integer programming to match formulas in an optimal fashion. Consequently, this method not only computes a general code similarity score as prior efforts [19, 35] do, but also presents in-depth matching results as evidence to justify the discovered bugs. In addition to search for security bugs in binary code based upon their distinctive contextual semantics, we also take a step further to enable human analysts to examine the searching results by producing an explainable diagnosis report that pinpoints the revealed bug code within the target program.

We have implemented a prototype, XMATCH, and systematically evaluated its performance with existing baseline methods using 72 binaries including 216,000 functions from x86 and MIPS. Experimental results have shown that XMATCH can further improve the search accuracy of existing bug search techniques with very reasonable performance overhead. XMATCH gets average recall rate 0.87 at top 1 for small functions (with the number of basic blocks less than 9). Furthermore, XMATCH can precisely locate well known vulnerabilities at top 1 in the real-world firmware image [3].

Contributions. In summary, the contributions of this paper are as follows:

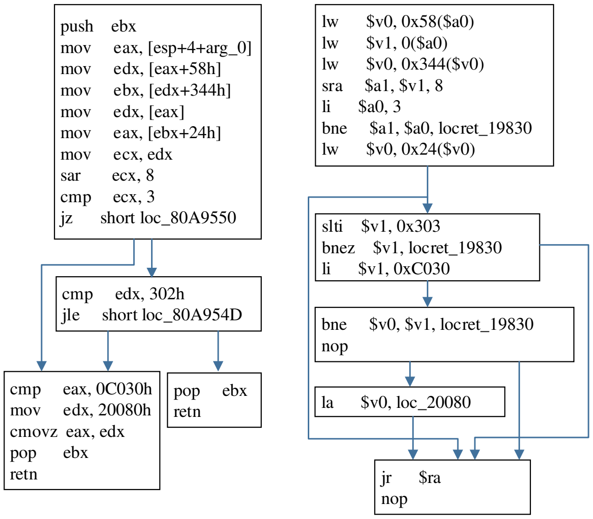
- We propose to extract conditional formulas as semantic features for cross-platform binary code search in binaries. Compared to prior work, a conditional formula distills contextual semantics (including triggering conditions and data-flow relations), therefore leading to higher search accuracy and providing explainable evidence for human inspection.
- We have implemented a prototype system, XMATCH, and systematically demonstrated the effectiveness of XMATCH on real-world samples including well-know libraries and the vulnerable firmware. The experiment shows that XMATCH can greatly improve the search accuracy of existing bug search techniques in the cross-architecture setting. Furthermore, it can precisely locate well known vulnerabilities at top 1 in the real-world firmware image where existing approaches failed for most of vulnerabilities. We also utilizes 5 case studies to show how XMATCH facilitates the security evaluator by providing the explanatory search results.

2. APPROACH OVERVIEW

2.1 Problem Statement

Our use scenario is similar to the previous work [19, 20, 35]. Given a vulnerable binary code function in one architecture, we search for the presence of the same function in other binaries in different architectures. In addition to ranking candidate functions based on their similarity scores, we like to provide explainable matching evidence for human analysts to eliminate false positives and determine if a matched function has been patched or not.

Motivating Example. We use a real-world vulnerability CVE-2013-6449 as a motivating example. Figure 1 illustrates two binary functions of `ssl_get_algorithm2` for x86 and MIPS. The x86 version is compiled from the OpenSSL library (version 1.0.1a), and the MIPS version is directly extracted from the Linux-based router firmware DD-WRT (version r21676) [3]. Both code snippets contain the vulnerability *CVE-2013-6449*, which allows remote attackers to launch a denial-of-service attack (daemon crash) via crafted traffic from a TLS 1.2 client. In this example, the vulnerable function `ssl_get_algorithm2` for x86 is provided to us, we aim to search the same vulnerable function in the stripped binaries from the router firmware, and from the matching results determine if each matched candidate is indeed vulnerable: it is neither a false positive or a patch that has been applied to the vulnerable function.



```
push   ebx
mov    eax, [esp+4+arg_0]
mov    edx, [eax+58h]
mov    ebx, [edx+344h]
mov    edx, [eax]
mov    eax, [ebx+24h]
mov    ecx, edx
sar    ecx, 8
cmp    ecx, 3
jz     short loc_80A9550

cmp    edx, 302h
jle    short loc_80A954D

cmp    eax, 0C030h
mov    edx, 20080h
cmovz  eax, edx
pop    ebx
retn

pop    ebx
retn
```

```
lw     $v0, 0x58($a0)
lw     $v1, 0($a0)
lw     $v0, 0x344($v0)
sra    $a1, $v1, 8
li     $a0, 3
bne    $a1, $a0, locret_19830
lw     $v0, 0x24($v0)

slti   $v1, 0x303
bnez   $v1, locret_19830
li     $v1, 0xC030

bne    $v0, $v1, locret_19830
nop

la     $v0, loc_20080

jr     $ra
nop
```

a) x86 assembly **b) MIPS assembly**

Figure 1: The control flow graph comparison for the vulnerable function `ssl_get_algorithm2` (*CVE-2013-6449*) under different architectures(x86 vs. MIPS).

Challenges. From this example, we observe several challenges for cross-platform binary code search:

- **Syntactic representations are very different.** x86 and MIPS have drastically different instruction sets. They employ different strategies to pass function arguments: x86 often utilizes the stack to pass arguments, whereas MIPS holds arguments in special registers. Moreover, they use different mechanisms for conditional branching: x86 relies on an implicit `EFLAGS` register, while MIPS does not. As we can see in Figure 1, the instruction syntaxes and the instruction counts are vastly different. Therefore, any code search techniques based on superficial features (e.g., opcode type and amount) might not produce very good accuracy.
- **Control-flow graphs are inconsistent.** The control-flow gra phs of the same function compiled on two platforms bear very different structures. In Figure 1, while x86 binary contains 4 basic blocks, MIPS function has 5 of them. Such CFG changes might become significant obstacles for previous code search efforts [19, 35], which rely on basic-block level feature extraction and semantics comparison. In contrast, to address this problem, we propose to search security bugs using the higher-level behavior-based semantics, including data dependency and branch predicate. These factors reveal the fundamental program behavior rather than volatile code formation, and therefore are insensitive to CFG-level structural variation.
- **Vulnerable code logic often is scattered across multiple basic blocks.** The vulnerability shown in Figure 1 is caused by an incorrect version check on the argument of `ssl_get_algorithm2`, but this code logic spans over multiple basic blocks and is blended with other code logics. Therefore, to precisely locate and confirm this vulnerability, it is insufficient to match individual basic blocks, as prior work [35] did. Instead, it is necessary to consider the vulnerability, involving multiple basic blocks, as an entirety and reconstruct

$$\frac{(([[a0]]/0x10 \mathrel{!=} 0x3) \parallel ([[a0]] < 0x302) \parallel ([[[[a0] + 0x58] + 0x344] + 0x24] \mathrel{!=} 0xc030))}{ret = [[[[a0] + 0x58] + 0x344] + 0x24]} \downarrow$$

$$\frac{(([[a0]]/0x10 == 0x3)\ \&\&\ ([[a0]] >= 0x302)\ \&\&\ ([[[[a0] + 0x58] + 0x344] + 0x24] == 0xc030))}{ret = 0x20080} \downarrow$$

a) ssl_get_algorithm2 **in x86**

$$\frac{(([[a0]]/0x10 \mathrel{!=} 0x3) \parallel ([[a0]] < 0x303) \parallel ([[[[a0] + 0x58] + 0x344] + 0x24] \mathrel{!=} 0xc030))}{ret=[[[[a0] + 0x58] + 0x344] + 0x24]} \downarrow$$

$$\frac{(([[a0]]/0x10 == 0x3)\ \&\&\ ([[a0]] >= 0x303)\ \&\&\ ([[[[a0] + 0x58] + 0x344] + 0x24] == 0xc030))}{ret = 0x20080} \downarrow$$

b) ssl_get_algorithm2 **in MIPS**

Figure 2: Conditional Formulas for the Motivating Example

high-level code logic that includes both the data and control dependency relations across basic blocks.

2.2 Conditional Formula

Definition 1. A *conditional formula* (CF) consists of an *action* (which describes how a function output is computed from one or more function inputs), and an optional *condition* (which is a boolean expression that triggers the execution of the formula).

```
⟨CondFormula⟩  ::= ⟨Action⟩
                 |  ⟨Condition⟩ '→' ⟨Action⟩
⟨Condition⟩    ::= ⟨Expression⟩
⟨Action⟩       ::= ⟨Assignment⟩
                 |  ⟨Function⟩
⟨Function⟩     ::= ⟨FnName⟩ '(' ⟨ParamList⟩ ')'
⟨ParamList⟩    ::= ⟨Expression⟩
                 |  ⟨Expression⟩ ',' ⟨ParamList⟩
⟨Assignment⟩   ::= ⟨Expression⟩ '=' ⟨Expression⟩
⟨Expression⟩   ::= ⟨Name⟩
                 |  ⟨Number⟩
                 |  ⟨Function⟩
                 |  ⟨Expression⟩ ⟨BinOp⟩ ⟨Expression⟩
                 |  '(' ⟨Expression⟩ ')'
                 |  '[' ⟨Expression⟩ ']'
```

Figure 3: Abbreviated BNF for Conditional Formula

An Backus-Naur Form for conditional formula is shown in Figure 3. A conditional formula `<CondFormula>` consists of a condition expression `<Condition>` with an action statement `<Action>`, with a connector → in between. If the triggering condition is always true, `<CondFormula>` is simply `<Action>`. While a condition `<Condition>` is a standard expression `<Expression>`, an action `<Action>` can be either an assignment `<Assignment>` or a function call `<Function>`. An expression can be as simple as an integer number, a variable name, a function call, a combination of two subexpressions connected by a binary operation (such as '+', '-', '&&', etc.), or surrounded by a pair of parentheses '()' or square brackets '[]'. Note that a pair of square brackets '[]' denote a memory dereference.

Figure 2 shows the conditional formulas for the two binary functions in Figure 1. Given the vulnerable `ssl_get_algorithm2` in x86, we can precisely label the vulnerable logic, the invalid condition check on its conditional formulas. XMATCH searches the whole `OpenSSL` binary in DD-WRT and finds a candidate, `ssl_get_algorithm2` in MIPS. It shares the similar code logic. XMATCH also produces the best match between the conditional formulas in both functions. As we see, the code logic is interpreted by the conditional formulas, and it becomes evident for an analyst to diagnose the vulnerability in DD-WRT firmware.

2.3 Workflow

We outline our approach in Figure 4. It consists of the following three steps: binary lifting, conditional formula extraction, and conditional formula matching.

Binary Lifting. We first utilize binary lifting to convert different native machine code to the same higher-level intermediate representation (IR). The lifted binary retains semantics that are consistent with the original binary program. Our subsequent operations will be directly conducted on the lifted binary.

Conditional Formula Extraction. We apply the binary analysis techniques on the lifted binary to construct conditional formulas. We carefully handle the data dependency via pointers. Besides, not all the variables in a lifted binary function are of interests. We conduct the action point selection to filter irrelevant variables.

Conditional Formula Matching. We match functions by their unified conditional formulas. We model such a matching problem as a linear assignment problem and leverage integer programming techniques to find an optimal solution. The matching result is then a one-to-one mapping of CFs, in addition to a simple similarity score. Human analysts can thus inspect the in-depth mapping results to understand and verify any discovered bugs.

3. BINARY LIFTING

Binary lifting transforms binary code of different architectures into a common code representation to facilitate subsequent analyses. We need to extract conditional formulas for one function, such a transformation must preserve the semantics of the entire function. To do so, we first recover the control flow graph of a function, and then transform the binary code instruction by instruction following the control flow graph.

With respect to the implementation, our binary lifting is based on McSema [14], a code translation framework that translates x86 instructions to LLVM IR (Intermediate Representation). To address the problem of cross-platform bug search, we have extended McSema in two fronts: 1) multi-architecture support; and 2) function prototype based translation.

3.1 Support for Multiple Architectures

McSema only supports translation from x86 instructions to LLVM IR. In our use scenario, we would like to translate binary code from

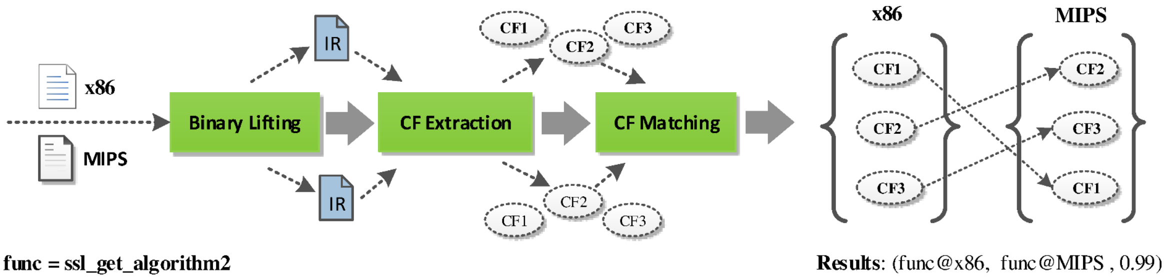


Figure 4: The overview of our approach. The inputs are binaries implementing the `ssl_get_algorithm2` function for x86 and MIPS, which contains the vulnerability *CVE-2013-6449*. First, the two binaries are lifted into an intermediary representation (IR). Second, conditional formulas are extracted from the lifted binary function. Finally, the conditional matching works for the similarity score, and one-to-one mapping results are outputted.

a wide variety of CPU architectures into LLVM IR. Therefore, we have to extend its support for other architectures. Fortunately, McSema provides a generic framework enabling us to easily support other instruction sets. In the current implementation of XMATCH, we extended its support for MIPS, since it is the popular CPU architecture for embedded systems and IoT devices.

McSema requires two steps to translate a binary function: 1) control flow graph recovery, and 2) bitcode generation. The control flow-graph recovery will disassemble a binary function, retrieve its basic blocks as well as control flow dependencies among these basic blocks. The bitcode generation walks through the control flow graph, conducts the one-by-one instruction translation and generates the LLVM bitcode file. We utilize IDA Pro to retrieve the control flow graph for a MIPS binary function. McSema translates each instruction in a x86 binary by modeling its execution semantic. We follow the similar instruction translation process. MIPS belongs to RISC instruction sets, so the amount of work for adding such support in McSema is much simpler than that of adding support for CISC instruction sets like x86. In our case, we add less than 1K LOC in McSema, and it is one-time effort work.

3.2 Function Prototype Based Translation

For function call translation, McSema introduces a global "context registers" data type and uses this as the only argument for all lifted functions. "context register" includes all the registers under corresponding CPU architecture. At the beginning of a lifted function, McSema first allocates several local variables, and spills all the registers in the global "context register" argument to those variables. Then the following operations are performed based on the variables. When a function returns, the "context register" will be wrapped up with the latest variables value. At each function call site, which means a lifted function would be called, the "context register" is first wrapped up, and then is passed to the callee as the only argument. As a result, McSema can preserve the control and data flow dependencies among functions without the need of function prototype recovery. However, this translation strategy will undermine the efficacy of XMATCH. Firstly, the generated condition formula fails to represent execution semantics on the real argument of a function without the function prototype recovery, since some flaw code logic could be related to the real argument on a function call such as memcpy. Secondly, unified function prototypes will reduce the accuracy of XMATCH, since we cannot rely on the number of arguments to further refine the search result.

To address these issues discussed above, we require McSema to translate function call instruction based on the function prototype. More specifically, there are three steps to achieve this goal: the function prototype recovery, the function call translation, and argument passing modeling. We first recover the function prototype for a binary function, and modify the function call translation mechanism in McSema based on recovered function prototype. We also add additional IR instructions to model the argument passing to the corresponding callsite.

The function prototype includes function name, its arguments and return values. We utilize IDA pro to obtain the recovered function prototypes. Compared with many other function prototype algorithms and platforms [10, 18, 31], IDA pro could have the limitation. However, it is user friendly and supports multiple platform. Besides, our experiments show that it is enough for our experiment purpose. In the future, we will adopt more robust approach for the further improvement. We assign all functions on the function callsite with a return value during the translation. We will take the screening process in Section 4.1 to reduce the impact of fake return value on the generated conditional formulas.

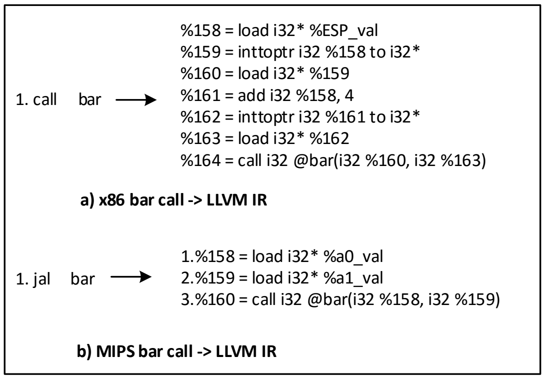
When McSema translates the function call instruction, it will predefine the number of arguments to translate. In its original design, it always assumes the number of argument to be 1. In our scenario, the number of arguments is defined by the recovered function prototype. We create argument variables with the same number of arguments defined in the function prototype. We also create the return variable for each lifted function.

Since the function call instruction translation has been changed in McSema, we also need to add corresponding argument passing instructions to preserve the data-flow dependency between the caller function and its callee. Modeling the argument passing depends on the calling convention type of the original binary function. We model calling conventions by their types, and check the calling convention type by matching our modeled patterns. With the calling convention type, we add corresponding argument passing IR instructions before the function call instruction.

Figure 5 shows a concrete example of how to translate a call on both x86 and MIPS architecture. In this example, from the analysis in IDA, we know that the function `bar` has two parameters, so at the call site, `call bar` and `jal bar` will be translated into a number of IR instructions shown on the right hand side of the figure. Instructions before the call instruction describe how arguments are passed into the corresponding function.

3.3 Other Issues

McSema does not support translation for all sorts of x86 instructions. For instance, it only supports a small portion of floating point instructions. However, McSema is well documented and it is not difficult to add support for other instructions that needed. In our



```
1. call    bar  ⟶   %158 = load i32* %ESP_val
                    %159 = inttoptr i32 %158 to i32*
                    %160 = load i32* %159
                    %161 = add i32 %158, 4
                    %162 = inttoptr i32 %161 to i32*
                    %163 = load i32* %162
                    %164 = call i32 @bar(i32 %160, i32 %163)
```

```
1. jal    bar  ⟶   1.%158 = load i32* %a0_val
                   2.%159 = load i32* %a1_val
                   3.%160 = call i32 @bar(i32 %158, i32 %159)
```

Figure 5: The example of the code translation for the `call` instruction.

case, it is important for us to provide support for the conditional branches which is necessary to conditional formula extraction, so we add support this kind of floating point instructions that McSema has not supported. We believe that supporting all instructions is engineering work and leave it as future work.

4. CONDITIONAL FORMULA EXTRACTION

We conduct static analysis directly on the lifted function to extract its conditional formulas. More specifically, we conduct intra-procedural dataflow analysis to construct formulas for *actions*, and perform the path slicing to retrieve the corresponding *conditions*. All static analysis are conducted on top of LLVM framework [1].

4.1 Action Construction

An *action* is a data-flow equation on a specific IR variable, which serves as a function output. To construct an action, we first discover all the function outputs (we call them "action points") that have external impacts. Then, starting from each action point, we compute the use-def chain to evaluate the reachability from function output to inputs. Eventually, we fold all the IR statements on each trace to produce an data-flow equation as an action for the corresponding action point.

Action Point Selection. Intuitively, we can compute data-flow equations for any IR variables hosted in a function. However, the lifted function still keep many architecture specific variables, such as `ESP_val` in Fig. 5a) and `a0_val` in Fig. 5b). This will make the generated conditional formulas to be dramatically different across architectures. Therefore, we only focus on the stable output states, which indicate consistent program behavior. To this end, we aim to calculate backward data-flow from three types of function outputs: 1) return values, 2) memory variable, 3) function calls.

1) **Return value.** We conduct a conservative analysis to identify the variables that hold return values, even if we assume all functions in a lifted binary have the return values during the binary lifting process discussed in Section 3.2. First, we seek IR variables with specific register names. This is due to the fact that a certain architecture uses specific registers to hold return values and IR variables, though lifted from binary, still preserve the original register names. Second, among these candidate variables, we further search for those that are never redefined in the same function. We then consider these variable to contain the return values.
2) **Memory variables.** A function can also write to memory. This is translated into a memory write operation in lifted binary. Therefore, we obtain memory variables by first searching for memory write instruction $storeinst$ in LLVM-IR. Next, we perform value-set analysis [7] to determine the memory region a pointer points to. Once a memory region is not updated in a function, its pointer is now pointing to the actual output, and thus can be considered as an action point.
3) **Function calls.** A function may call another function. If the caller function uses or checks the return value of the callee function, then the callee function will be eventually included in the data-flow expression of the variable that uses the return value. If the return value of the callee function is never used inside the caller function, we will treat it as an action point.

Data-Flow Analysis. Once we have discovered all the action points, we then compute backward data-flow for each one of them. To this end, we first perform use-def chain analysis. The use-def chain analysis needs to consider two types of variables in a lifted binary: register variables and memory variables. LLVM IR is already in SSA (Static Single Assignment) form, so we can directly retrieve the use-def chain for register variables. However, memory variables are address-taken variables which are not in SSA form in LLVM IR. We need promote memory variables to register variables first to obtain their use-def chains.

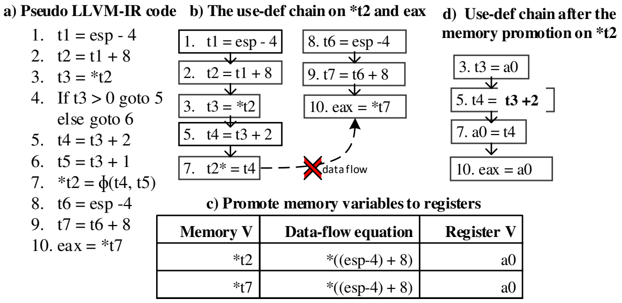
Memory promotion is to promotes memory references to be register references. It firstly collects all possible memory variables by finding pointers holding their addresses. The pointer variable is transformed to a register variable by rewriting the lifted IR function, then traversing the function in depth-first order to rewrite all its uses as appropriate. This just follows the standard SSA construction algorithm.

The binary lifting translates the pointer variable by using "inttoptr" instruction. We find all potential pointer variables by scanning the whole binary function for this instruction. The next step is to rewrite the pointer as well as its all uses. We need conduct the value-set analysis [7] on pointer variables to find all its uses. Since the memory variable is not in SSA form, it is hard for us to know which pointers share the same value. To this end, each pointer is labeled by its address, the a-loc (known as "abstract location") of the memory region that it accesses. Hence, the reaching-definition analysis on the pointer variable aims to track its `Kill` and `Gen` sets of a-locs. The result of this analysis in effect recovers the data-flow dependencies among memory regions that are accessed using pointers, and therefore help bridge the disconnected data-flows. After recovering the data-flows among pointer variables, we convert them into SSA form by the standard SSA construction algorithm.

We utilize the data-flow expression on the index of a pointer variable to determine its a-loc. This is also widely adopted in other works [6]. The data-flow expression on the index describes how the index is computed. Pointers of the same memory variable should share the similar data-flow expression. We utilize a theorem prover [2] to further unify the data-flow expression for a-loc.

There are three types of pointers: global pointers, local pointers and pointers from function arguments. The a-loc of a global pointer is a constant address which is different cross architectures. We assign each of global variables, by orders of their addresses, a new variable "GN", where "N" is the its order. We also follow the same rule to rename the local pointers into "LN", and pointers from arguments into "a0" to "aN".

We will rewrite the IR function by renaming all pointer variables according to their a-locs, and by traversing the function in depth-first order to rewrite all its uses as appropriate. The result IR function has the complete use-def chain for both register variables and memory variables.



```
1.  t1 = esp - 4
2.  t2 = t1 + 8
3.  t3 = *t2
4.  If t3 > 0 goto 5
    else goto 6
5.  t4 = t3 + 2
6.  t5 = t3 + 1
7.  *t2 = ϕ(t4, t5)
8.  t6 = esp -4
9.  t7 = t6 + 8
10. eax = *t7
```

Memory V	Data-flow equation	Register V
*t2	*((esp-4) + 8)	a0
*t7	*((esp-4) + 8)	a0

Figure 6: The example of lifted IR code and generated statements for its variables.

Action Generation. The action generation works on the rewritten IR function. Starting from an action point, we fold all the instructions on every single data-flow path and produce a data-flow equation. One such equation is further simplified using theorem prover and the result is then considered to be one action.

Running Example. Figure 6 illustrates the action construction process. For readability purpose, we utilize pseudo code instead of LLVM-IR in the demonstration. Figure 6(a) shows the IR in SSA (Static Single Assignment) form. The action point in this example is `eax`, which holds the return value. Figure 6(b) shows the incomplete use-def chain without the memory variable promotion on `*t2` and `*t7`. We can see that the data flow between `eax` and `t4` is missing, so we cannot know the current value of `eax` is `[esp+4]+2`. Figure 6(c) and (d) illustrates the effectiveness of memory variable promotion. By analyzing data-flow equation on pointers `*t2` and `*t6`, we identify that these two pointers access the same a-loc `a0`. Such a discovery helps us further connect the use-def chain from eax to `*t2`. Figure 6(d) demonstrates the updated use-def chain by considering pointers. Eventually, we compute two data-flow equations for the action point eax: `a0+1` and `a0+2`, each of which is generated from one single path.

a) Action

Variable	Action
eax	a0 + 2
	a0 + 1

c) Conditional formula

Action	Condition	Value
a0+2	a0 > 0	True
a0+1	a0 > 0	False

Action	(a0+2)	(a0+1)
Slicing criterion	(eax=a0,{a0, t4, t3})	(eax=a0,{a0, t5, t3})
Path slice	3. t3 = a0 4. If t3 > 0 goto 5 else goto 6 5. t4 = a0 + 2 7. a0 = t4 10. eax = a0	3. t3 = a0 4. If t3 > 0 goto 5 else goto 6 6. t5 = a0 + 1 7. a0 = t5 10. eax = a0

b)slicing criterion and sliced paths

Figure 7: The condition generation for IR variable `eax`.

4.2 Condition Extraction

We further utilize the path slicing algorithm [26] to generate conditions for each action. In our scenario, the path slicing is used to extract conditions for the specific path holding the action in the lifted binary function.

Given the action and the computed data-flow, we set the slicing criterion to include all variables in the action. With the slicing criterion, the path slicing algorithm will trace backwards to find the path slice which contains all the variables in the slicing criterion. We extract all the comparison variables from the path slice and generate predicate expressions for these variables. Each predicate expression includes the condition expression and its boolean value that will cause the action to be executed. In LLVM-IR, the comparison variable is the first operand of the branch instruction, if it is the conditional jump. We generate the data-flow equation for the condition variable to get the condition expression on this basic block. We can obtain the boolean value by checking the successor block on the path. If its address is the second operand of the branch variable, the boolean value is `true`. Otherwise, `false`. If the boolean value is false, we will negate the condition expression. We make a conjunction of the discovered predicate expressions on the path slice as the condition for the action v.

Running Example. Figure 7 demonstrates the condition generation process for actions of `eax`. Figure 7(a) lists two actions for `eax`: `a0+2` and `a0+1`. This indicates that the eax may hold two different values depending on which data-flow path to take. Figure 7(b) shows the path criterion of the two actions, each of which involves 3 IR variables. The corresponding path slices for the action are also presented in Figure 7(b). This slices include not only the data-flow but also all the conditions. We then can walk through these path slices, extract all branch variables and make a conjunction of their data-flow equations. The result becomes the condition for this action. The outputs of condition generation for action `a0+1` and `a0+2` are listed in Figure 7(c).

Arch	Condition Expression
X86	((%t3-0)==0, ((%t3-0)<0) == and(lshr(and (xor([esp-8],3),xor(xor([esp-8],3)), (%t3-0))),31),1)))
MIPS	(%t3-0)==0

Figure 8: The demo example for condition expression in x86.

Architecture-specific Conditional Expression. For some architectures, such as x86, we cannot directly use the generated condition for comparison, since its conditions are data equations on status registers. As a result, the condition expression is completely different from those in other architectures, such MIPS. For example, Figure 8 shows the condition expression for comparison variable $\%t3$ in x86 and MIPS. We address this issue by building the model for each condition expressions on status registers. Then we map them into corresponding real conditional expressions.

5. CONDITIONAL FORMULA MATCHING

We match two functions by their conditional formulas. It consists of two steps. Firstly, we compute the matching cost of two *CF*s. Secondly, we seek the optimal matching among *CF*s by selecting the minimum matching costs between two sets of *CF*s, and then output the similarity score of two functions.

Intuitively, one can utilize string edit distance to calculate the match cost between two *CF*s. However, two semantically equivalent *CF*s may appear to be different due to different ordering. For instance, **((a>0) && (b>0))** / *ret = 0x20800* will be considered unequal to **((b>0) && (a>0))** / *ret = 0x20800*, even though they share the same behavior-level semantics because of the commutative property. To avoid the reordering problem, we instead match two *CF*s by their AST structure. Since the AST is a tree-like structure, we adopt the graph edit distance to calculate the cost to transform between two *CF*s.

We utilize the algorithm [38] to compute the graph edit distance $ged(cf_i, cf_j)$. In our case, not all nodes are inter-changeable. For example, the condition related node cannot be replaced by action related node. Therefore, in the pre-computed mapping cost matrix in the algorithm, we assign the infinite number for the mapping cost between action and condition node.

The distance of two CFs is then used to calculate the match cost of two functions. Suppose we are given two functions f_1 and f_2, where f_1 contains the CF set $\{cf_1, cf_2, \ldots, cf_n\}$ and f_2 holds the set $\{\hat{cf}_1, \hat{cf}_2, \ldots, \hat{cf}_m\}$. Let $\mathbf{m}_{ij}$ be the matching factor for a CF pair: if cf_i matches $\hat{cf}_j$, $\mathbf{m}_{ij} = 1$; otherwise, $\mathbf{m}_{ij} = 0$. Hence, all the matching factors form a matching matrix $\mathbf{M}_{n\times m}$, which demonstrates how these two functions correspond to each other.

Following the graph edit distance, we define the *function distance* as the minimum distance of all matched CF pairs between f_1 and f_2. As we see, finding the function distance is equivalent to finding the $\mathbf{M}$ that minimizes $\sum_{(i,j)\in\{\mathbf{m}_{i,j}=1\}} dist(cf_i, \hat{cf}_j)$. In other words, we need to find the best match (with the minimum distance) between two functions. Note the greedy approach, which matches the CF in each function individually, cannot be applied as it can only produce the suboptimal solutions. To find the global optimal solution, we formulate the following objective function:

$$\begin{aligned} &\textbf{min} && \sum_{i=1}^{n}\sum_{j=1}^{m} \mathbf{m}_{ij} \times dist(cf_i, \hat{cf}_j) \\ &\textbf{subject to} && \sum_{i=1}^{n} \mathbf{m}_{ij} = 1, \forall j \in \{1, m\} \\ &&& \sum_{j=1}^{m} \mathbf{m}_{ij} = 1, \forall i \in \{1, n\} \\ &&& \mathbf{m}_{ij} \in \{0,1\} \forall i \in \{1,n\}, \forall j \in \{1,m\}. \end{aligned} \tag{1}$$

The objective in Eq. (1) is to calculate the distance between matched CF pairs, and minimize this value. The constraints indicate every CF in the functions can be matched only once. Based on Eq. (1), we can formally introduce the function distance.

Definition 2. The function distance of f_1 and f_2 is the minimum distance of all matched CF pairs between f_1 and f_2. Let $\mathbf{M}^*$ represent the optimal solution of Eq. (1), i.e. the best match between the two functions, the distance is calculated from:

$$dist(f_1, f_2) = \sum_{i=1}^{n}\sum_{j=1}^{m} \mathbf{m}^*_{ij} \times dist(cf_i, \hat{cf}_j), \tag{2}$$

where $\mathbf{m}^*_{ij}$ is an element in the optimal solution $\mathbf{M}^*$.

The function distance provides a finer-grained comparison for two functions. It not only quantifies the dissimilarity between two functions (minimum distance transforming one set of CFs into the other), but also explains how the statements of the two functions are matched together. The best match is stored in $\mathbf{M}^*$, where for all $\mathbf{m}^*_{ij} = 1$, we can plot a best match between the statement c_i in f_1 and the statement c_j in f_2. By tracking the conditional formulas in a vulnerable function, this matching helps human analysts locate potential vulnerable statements in other functions. We will elaborate this point in our experiments.

According to Definition 2, to calculate the function distance, we need to find the optimal solution of Eq. (1). Fortunately, the problem in Eq. (1) is a well studied problem called integer linear programming which can be efficiently solved by various techniques such as constraint relaxation [22, 43]. In this paper, we leverage the solution in [30] to solve the problem. Our experimental results show that the employed matching algorithm is efficient.

The function distance quantifies the difference between two functions which could be roughly proportional to the sum of the functions sizes (in bytes). Thus, two larger functions are likely to be more dissimilar. To reduce the bias, we normalize this absolute distance by the total length of conditional formulas in the two compared functions. We define the *function similarity* of f_1 and f_2 as:

$$sim(f_1, f_2) = 1 - \frac{dist(f_1, f_2)}{\sum_{i=1}^{n} len(cf_i) + \sum_{j=1}^{m} len(\hat{cf}_j)}, \tag{3}$$

where len counts the string length of each conditional formula. Its enumerator is the function edit distance and its denominator denotes the largest possible string edit distance between two completely different functions.

6. EXPERIMENTAL EVALUATION

In this section, we evaluate XMATCH with respect to its accuracy, explainability and runtime performance. First, we systematically compare the performance of XMATCH against existing baseline methods under the cross-platform setting (Section 6.2). Second, we apply XMATCH and baseline methods to detecting real-world vulnerable code snippets (Section 6.3). Then, we evaluate the precision of XMATCH via matching vulnerable code with patched ones (Section 6.4). Further, we demonstrate the explainability of XMATCH (Section 6.5). In the end, we measure the runtime performance of our tool (Section 6.6).

Table 1: The vulnerabilities used in our experiments.

Codebase	Function	Type
OpenSSL	EVP_DecodeUpdate	CVE-2015-0292
	X509_cmp_time	CVE-2015-1789
	dtls1 process heartbeat	CVE-2014-0160
	tls_decrypt_ticket	CVE-2014-3567
	dtls1_buffer_record	CVE-2015-0206
	X509_verify	CVE-2014-8275
	c2i_ASN1_OBJECT	CVE-2014-3508
	ssl_get_algorithm2	CVE-2013-6449
BusyBox	make_device	CVE-2013-1813
	xmalloc_optname_optval	CVE-2011-2716

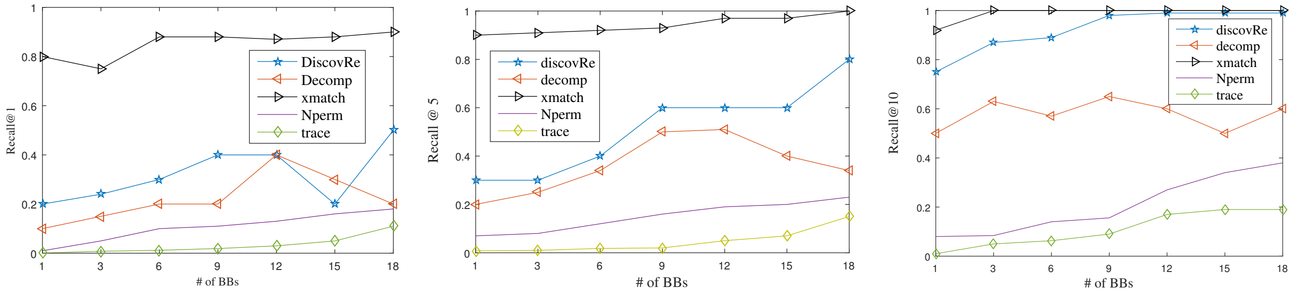
6.1 Experiment Setup

All experiments have been conducted on a machine with an Intel(R) Core i5 @ 2.9GHz and 16 GB DDR3-RAM, running 64-bit Ubuntu 14.04. We compiled the source code of two typical software OpenSSL and BusyBox, which are widely used in the firmware of IoT devices, and perform code search on the generated binaries. Specifically, we have compiled two versions (1.0.1.a, 1.0.2.d) of OpenSSL and two revisions (1.19.0, 1.20.0) of BusyBox, both on two 32-bit architectures (x86 and MIPS), with three compilers (gcc v4.8.4, gcc v4.8.1 and clang v3.4), and across three major OSes (Windows, Linux and Mac OS X). Thus, we have created 72 binaries in total as the baseline dataset. All the code searching experiments were conducted on the dataset, and we kept their debug symbols because they provide the ground truth to enable us to verify the correctness of matched functions. We also selected 10 representative vulnerabilities for evaluation, including the notorious HeartBleed bug.

6.2 Cross-Platform Baseline Comparison

To demonstrate the efficacy of XMATCH in terms of cross-platform code search, we compare our system with baseline methods.

Preparation of Baseline Systems. We have prepared 4 baseline systems for the comparative experiments. They include the state-of-the-art cross-architecture bug search technique discovRe [19], a decompiler-based approach, the n-gram based permutation-resistant matching technique Nperm [28] and the tracelet-based method [13]. Notice that, although Nperm and tracelet-based methods are not designed for cross-platform code matching, their techniques can be



(a) Recall@1

(b) Recall@5

(c) Recall@10

Figure 9: The cross-platform baseline recall comparisons under different function sizes. Recall@k means that the recall rate if we consider top k candidates as positives.

applied to cross-architecture setting once a binary is lifted to a uniformed code representation (i.e., intermediate representation).

- *discovRe*: Due to unavailability of the source code, we have re-implemented discovRe[1] and set the iteration limitation to be the same (i.e., 10,000) as the one used in the prior work.
- *Decomp*: We have also implemented a decompiler-based bug search system `Decomp`. It relies on the on-line retargetable decompiler service [4] to conduct decompilation and leverage a prior technique [25] to compute the similarity between two recovered C functions in order to perform code search.
- *Nperm*: We employed the "*N*-perms" technique in lifted binaries. The length parameter N is set to be 3, as suggested in the previous work [29].
- *Tracelet*: Similarly, we applied tracelet-based method to lifted binaries. Besides, to facilitate the matching process, we replaced the original optimization algorithm with a maximum value section. That is, we select the similarity score of two most similar traces as the one for the two functions.

Our baseline experiments mainly focuses on the function-level matching on the aforementioned 72 binaries. Given a function, we use each method to calculates its similarity scores against all functions in the binaries and produce a list of functions sorted in descending order of the scores. We disable function inlining during compilation since XMATCH does not currently support inclined code. Notice that this is a common limitation also shared by prior CFG-based approach [19].

Metrics. We used recall rate to evaluate the performance of the proposed and baseline methods. It is a standard evaluation metric, which calculates the fraction of correctly matched functions in the top-k retrieved instances. Here, k means we consider top k candidates as positives. Intuitively, a larger k leads to a higher recall for every method.

Comparison Results. We conducted two sets of experiments for baseline comparison. Firstly, we investigated the performance of proposed and baseline methods when handling different sizes of functions. Secondly, we systematically compared XMATCH with baseline methods on a real-world library `OpenSSL`.

For the first experiment, we clustered functions of different names in our dataset by the number of basic blocks. We randomly selected 500 functions for each cluster i. For each function, we collect its x86 and MIPS versions compiled by gcc v4.8.4. As a result, we have a new data set C=$\{c_1, \ldots, c_i\}$ where c_i contains 1000 functions with same size i. For each function in cluster c_i, we search the MIPS version using its x86 version in c_i. We use Recall@K to measure its accuracy. We ran XMATCH and baseline methods in different clusters c_i and obtained the evaluation results for Recall@1, Recall@5 and Recall@10.

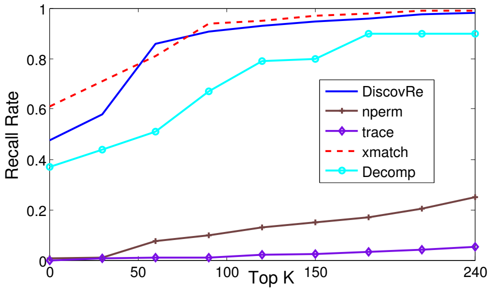


Figure 10: The cross-platform baseline comparison on 1,000 functions randomly selected from the dataset.

Figure 9 shows their matching accuracy for functions from size 1 to 18. In general, it demonstrates the accuracy of XMATCH is better than the state-of-the-art approaches. It is worth noting that the performance is evaluated 7,000 functions from the same benchmark, and the advantage of XMATCH over all other methods is statistically significant, according to a paired t-test (at the level $p = 0.05$).

Particularly, XMATCH outperforms all of the baseline methods for small functions (size$\leq$12). Figure 9 indeed justifies our argument: although discovRe is accurate for matching large functions whose basic block number is greater than 15, this technique is not favorable in searching small functions. This is because small functions have fewer constraints for CFG-based approaches to utilize. In contrast, XMATCH can achieve better accuracy due to the conditional formula based matching. Even small functions still have semantic-rich and thus unique conditional formulas. For instance, function `X509_verify` has only one basic block and therefore DiscovRe cannot rank it at recall@100. On the contrary, XMATCH can give it a top ranking because the enclosed conditional formulas are relatively unique.

In the second experiment, we randomly selected 1,000 functions without considering their sizes, and applied all mentioned approaches on this dataset. For each method, we also collected its recall rates for different threshold K. Figure. 10 shows the comparison results. We also can see that XMATCH can still outperform all baseline methods.

We also notice that the decompiler-based approach does not provide meaningful results in two sets of experiments. Most false pos-

[1] We contacted the author of discovRe to assist us by providing their search results, but they have not responded.

Table 2: The vulnerability ranking baseline comparison on DDWRT firmware and BusyBox

Software	Function Name	Decomp Rank	N-perm Rank	Trace Rank	DiscovRe Rank	Xmatch
OpenSSL1.01.a x86→ DDWRT(r21676)	dtls1_process_heartbeat	50	14	843	1*	1*
	EVP_DecodeUpdate	63	40	1569	10	1
	X509_cmp_time	266	4	1662	81	1
	tls_decrypt_ticket	90	7	1913	1	1
	X509_verify	844	413	540	125	1*
	c2i_ASN1_OBJECT	1100	624	689	3	1
	dtls1_buffer_record	574	7	1108	12	1
	ssl_get_algorithm2	59	230	1009	6	1
BusyBox1.19.0 x86 → BusyBox1.20.0 MIPS	make_device	180	69	4	6	1
	xmalloc_optname_optval	87	93	428	10	1

* means that there are multiple functions with the same similarity scores.

itives are caused by the decompilation errors. This justify the motivation of our approach.

We investigated false positives of XMATCH and found that most of them are caused due to two reasons. First, different functions share the similar code logics. We found that different functions may have similar code patterns except that they access different fields of the same object. This is common in the functions with small amount (e.g., only one) of basic blocks. We can handle such cases by further introducing context information, such as its callers and callees. Second, different functions share the same constants. Sometimes, identical constants such as object offsets are encoded into conditional formulas that are originated from binaries compiled with even different architectures. Such noises may dominate the similarity computation and lead to false matching. Nevertheless, XMATCH has already reduced false positive rate significantly, compared to baseline methods, due to its fundamental advantage of semantic and contextual awareness. Furthermore, the self-explanatory property of conditional formulas can facilitate further manual verification and help human experts easily screen these false positives.

6.3 Real-World Software Evaluation

To understand the effectiveness of our technique, we apply both XMATCH and the baseline methods to real-world vulnerable binaries. Specifically, we focus on 10 representative vulnerabilities in `OpenSSL` and `BusyBox`, each of which is corresponding to an individual function as presented in Table 1.

We further perform cross-platform search on a DD-WRT router firmware (r21676) [3]. Considering known vulnerable functions in `OpenSSL1.0.1a` binary (x86 using gcc 4.8.1) as the bug signatures, we search in `OpenSSL` binary from this DD-WRT firmware for the same bugs. Similarly, to uncover cross-platform vulnerabilities in `BusyBox` binary, we examine its MIPS distribution using bug signatures generated from x86 binary code. More concretely, we first compile an x86 version of `BusyBox1.19.0` using gcc 4.8.1; we create a MIPS version of `BusyBox1.20.0` using gcc 4.8.4. Then, we attempt to match the vulnerable functions from the former one with the unknown functions in the latter.

Table 2 illustrates the comparison results. XMATCH can always correctly discover the vulnerable functions as top candidates in the target binary. On the contrary, none of the baseline methods, including the state-of-the-art technique DiscovRe, can produce a high ranking for most of the buggy functions. For example, XMATCH can rank `X509_cmp_time` at top 1; on the contrary, discovRe can only rank it at top 81.

In the case of function `X509_verify`, aside from the true vulnerable function, XMATCH also identifies three other functions (e.g., `X509_REQ_verify`) to be top candidates because these functions all bear the same conditional formulas. By investigating these false positives in source code, we find that these "same" conditional formulas in fact access different types of data objects. However, due to the lack of high-level type information, XMATCH cannot distinguish such formulas from one to another. However, it is worth noting that these three functions may potentially be vulnerable since the presence of same conditional formulas could indicate the identical buggy program logics, which are left unpatched. We have contacted OpenSSL team to request further confirmation.

6.4 Unpatched versus Patched Code

One major challenge of bug search lies in that a patched version may have some differences with the original vulnerable function. Such difference may confuse XMATCH to get some false negatives. Thus, we would like to evaluate XMATCH with both buggy code and the corresponding patch in order to understand whether XMATCH can find the patched version or not. Furthermore, we hope to perform such a measurement in a cross-platform setting.

To this end, we first compiled a vulnerable `OpenSSL` (1.0.1a), with 5 representative bugs, using gcc 4.8.1 under both x86 and MIPS, and compiled a patched one (1.0.2d) using Clang 3.4 also under the same architectures. Then, we matched generated x86 binaries to MIPS ones. This involves four classes of matching: 1) patched-to-unpatched, 2) patched-to-patched, 3) unpatched-to-unpatched, and 4) unpatched-to-patched. Table 3 shows the results on 5 representative vulnerabilities. For each function, the matching result includes a candidate ranking and a similarity score.

Patched-to-Patched and Unpatched-to-Unpatched As a baseline, we first evaluated the matching between two patched or two unpatched binaries on different architectures. As depicted in Table 3, XMATCH can produce fairly high similarity scores (around 0.98 on average) between matched functions in these two classes. This again demonstrates that our conditional formulas can distill the essential program logics (vulnerable or not) while abstracting away the low-level architecture-specific details. However, we did notice that we cannot always exactly match the conditional formulas extracted from binaries in two different architectures and therefore cannot achieve a 1.00 similarity score. In a further investigation, we found that this imperfection is caused by the existence of global variables: indexes for global pointers used in conditional formulas may be different across architectures. We will address this issue in the further work by proposing a better way to index global pointers.

Patched-to-Unpatched and Unpatched-to-Patched For the function matching in these two classes, we noticed that a patched function is still considered as the top candidate of an unpatched one and

Table 3: Cross-platform patch code matching.(unpatched:OpenSSL 1.0.1a vs. patched:1.0.2d, x86 vs MIPS)

Vulnerable Function	P(x86) →U(MIPS)	P(x86)→P(MIPS)	U(x86)→U(MIPS)	U(x86)→P(MIPS)
dtls1_process_heartbeat	1/0.81	1/0.97	1/0.96	1/0.83
EVP_Decode_Update	1/0.90	1/0.99	1/0.99	1/0.88
X509_cmp_time	1/0.74	1/0.95	1/0.95	1/0.74
tls_decrypt_ticket	1/0.90	1/0.99	1/0.99	1/0.90
c2i_ASN1_OBJECT	1/0.66	1/0.99	1/0.99	1/0.66

P means patched version, and U means unpatched version. For each matching result `x/y`, `x` means ranking and `y` is similarity score.

vice versa. For instance, the patch for `tls_decrypt_ticket` consists of merely 3 lines of code and the one for c2i_ASN1_OBJECT, which is already fairly large, contains only 12 lines of code. In such cases, the rest of conditional formulas, which are left unchanged, may dominate the similarity computation.

6.5 The Case Study On Explainability

In this section, we demonstrate that the conditional formulas is able to provide self-explanatory evidence to human experts for further verification. In the paper, we will analyze one example as a demonstration. More examples can be found in Appendix.

To exemplify the intrinsic explainability of conditional formulas, we study the matching results from an unpatched x86 function to a patched MIPS version. Due to the page limit, we only demonstrate our analysis on the function `dtls1_process_heartbeat` whose unpatched version bears the Heartbleed bug (i.e., CVE-2014-0160). The analyses on other functions are presented in Appendix.

CVE-2014-0160 The result for the `dtls1_process_heartbeat` is shown in Figure. 11. The code snippet on the left represents conditional formulas extracted from patched MIPS and unpatched x86 binaries, respectively. The one on the right presents corresponding source code. The matched formulas are linked by red dotted lines.

The root cause of Heartbleed vulnerability is the missing length check for `memcpy()` arguments and such a check is introduced in the patch code. In contrast, the dataflow that reaches `memcpy()` remains intact. Both the modified condition and invariant dataflow can be directly observed from the two matched conditional formulas. Specifically, the If-clauses are different due to the introduction of new boundary check (depicted in bold), while the Then-clauses denoting `memcpy()` activities remain unchanged.

In this case, the identical and sophisticated Then-clauses indicate that the two functions are indeed very similar to one another. This explains why XMATCH considers the patched MIPS function to be the top matching candidate for a vulnerable one. Nevertheless, since two functions bear different behaviors in terms of condition check, their similarity score is relatively low (0.81).

In addition, due to the behavior level matching, XMATCH can explain that these two functions, buggy and patched, are corresponded to one another exactly because they share these similar conditional formulas. Thus, by assessing the difference between two *CF*s, which includes barely 2 predicates in the If-clauses, a human analyst can easily understand and rule out such a false alarm. In contrast, prior work can only output the similarity score and matched control flow graphs without pinpointing the exact matching regions. In that case, human experts will have no choice but to manually analyze the binary code to dig out the vulnerable logic for further verification.

6.6 Runtime Performance

We tested XMATCH on 1000 randomly selected functions from the dataset in Section 6.1 and evaluated the runtime in three steps, i.e. binary lifting, conditional formula extraction, and function matching. The first two steps are offline steps that can be preprocessed beforehand; whereas the last step is an online search step.

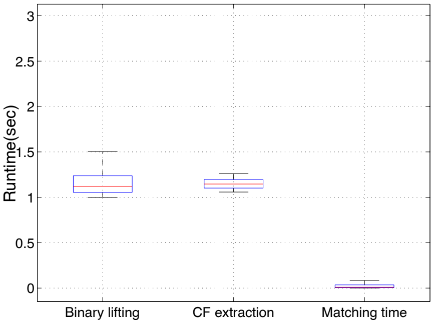


Figure 12: The runtime performance of XMATCH

The results are presented in Figure 12. On average, the binary lifting and conditional formula extraction take 2.3 seconds, and it takes 0.029 seconds to perform the function level matching. Overall, no matching takes longer than 0.55 seconds for all functions. The maximum preprocessing time (binary lifting and conditional formula extraction) is about 2.9 seconds. The preprocessing can be easily executed in parallel across multiple machines, and thus is not the bottleneck of our system. The search time grows linearly with the number of searched functions, and thus the function search is reasonably fast. We plan to further improve the performance of XMATCH by using the indexing techniques in our future work.

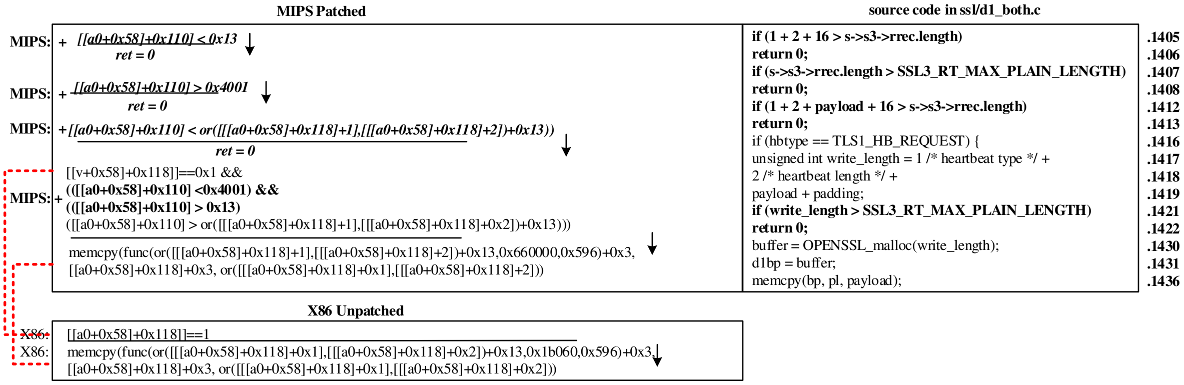
7. DISCUSSION

In this section, we discuss about the limitation and potential challenges of this work.

Loop Handling We only unroll the loop once during the data flow analysis. This is a safe strategy, because it does not increase the false negatives of the match result. Besides, this strategy is also attempted in other works [9], In the future, we could apply the existing loop analysis such as the technique [23] to further improve the accuracy of our approach.

Vulnerability across multiple functions. The goal of XMATCH is not to create the abstract formula cross multiple functions to find the potential vulnerabilities. If the vulnerability is related to multiple functions. XMATCH will find all related functions for the further vulnerability diagnosis.

Function Inlining. XMATCH can handle the inlined function which does not affect the most of code logics in the caller function. If the inlined function changes most of code logics in the caller function, we need to extend XMATCH to support inter-procedure analysis to not only generate the conditional formulas for one function but a set of functions. We will study how to systematically address this problem in future work.



MIPS Patched

```
MIPS: + [[a0+0x58]+0x110] < 0x13
              ret = 0
MIPS: + [[a0+0x58]+0x110] > 0x4001
              ret = 0
MIPS: +[[a0+0x58]+0x110] < or([[[a0+0x58]+0x118]+1],[[[a0+0x58]+0x118]+2])+0x13))
              ret = 0
        [[v+0x58]+0x118]]==0x1 &&
MIPS: + (([[a0+0x58]+0x110] <0x4001) &&
        (([[a0+0x58]+0x110] > 0x13)
        ([[a0+0x58]+0x110] > or([[[a0+0x58]+0x118]+1],[[[a0+0x58]+0x118]+0x2])+0x13)))
        memcpy(func(or([[[a0+0x58]+0x118]+1],[[[a0+0x58]+0x118]+2])+0x13,0x660000,0x596)+0x3,
        [[a0+0x58]+0x118]+0x3, or([[[a0+0x58]+0x118]+0x1],[[[a0+0x58]+0x118]+2]))
```

source code in ssl/d1_both.c

```
if (1 + 2 + 16 > s->s3->rrec.length)                        .1405
return 0;                                                   .1406
if (s->s3->rrec.length > SSL3_RT_MAX_PLAIN_LENGTH)          .1407
return 0;                                                   .1408
if (1 + 2 + payload + 16 > s->s3->rrec.length)              .1412
return 0;                                                   .1413
if (hbtype == TLS1_HB_REQUEST) {                            .1416
unsigned int write_length = 1 /* heartbeat type */ +        .1417
2 /* heartbeat length */ +                                  .1418
payload + padding;                                          .1419
if (write_length > SSL3_RT_MAX_PLAIN_LENGTH)                .1421
return 0;                                                   .1422
buffer = OPENSSL_malloc(write_length);                      .1430
d1bp = buffer;                                              .1431
memcpy(bp, pl, payload);                                    .1436
```

X86 Unpatched

```
~~X86:~~ [[a0+0x58]+0x118]]==1
X86:  memcpy(func(or([[[a0+0x58]+0x118]+0x1],[[[a0+0x58]+0x118]+0x2])+0x13,0x1b060,0x596)+0x3,
      [[a0+0x58]+0x118]+0x3, or([[[a0+0x58]+0x118]+0x1],[[[a0+0x58]+0x118]+0x2]))
```

Figure 11: The explainability demo for `dtls1_process_heartbeat` vulnerability

8. RELATED WORK

We have discussed closely related work throughout the paper. In this section, we briefly survey additional related work. We focus on approaches using code similarity to search for known bugs. There are many other approaches that aim at finding known bugs [25, 27, 32] in source code, or unknown bugs, such as fuzzing or symbolic execution [5, 11, 12, 37, 40, 42]. Since they are orthogonal to our approach, we will not discuss these approaches in this section.

Binary-only Code Search. Compared with source-code-level search techniques, binary-level code search is far more challenging. Most of these works do not focus on matching functions by their code logics. Instead, they focus on matching syntactic features [24, 29], semantic features [36], basic-block-level I/O dependencies [35], or code environments [17]. All these approaches only give the similarity scores. Therefore, they cannot provide a reasoning scheme which can also give effective evidence about why the target code is vulnerable. Tracelet-based approach [13] gives the accountable matching result, which shares the similar idea with ours. However, it cannot be applied on the lifted binaries. We have demonstrated this point in Section 6. Furthermore, BinHunt [21] and iBinHunt [33] utilize symbolic execution and a theorem prover to check semantic equivalence between basic blocks. These two approaches are expensive and cannot be applied for large scale firmware bug search, since they need to conduct binary analysis to extract the equations and conduct the equivalence checking.

Existing Binary Analysis Tools. In this paper, we do not invent new binary analysis techniques. Instead, we leverage the existing binary analysis techniques to extract conditional formulas for code search. Therefore, our proposed approach can be applied into more mature platforms such as BAP [8], Bitblaze [41], or Panda [15]. Besides, lifting binaries into the intermediate representation has been well studied. We choose LLVM-IR, because LLVM framework is mature and has many excellent optimization features. The binary lifting of the paper can also be implemented by other types of IR such as Valgrind VEX [34], BAP BIL [8], or REIL [16].

Decompilation Related Approach. Decompilation can provide more readable code, but that is not explainable. This is because it does not conduct the factorization on the function to extract independent code logics. An analysis still needs to manually check the text to locate the potentially vulnerable code logic. Program analysis on generated C code could facilitate this process, but the quality of decompiled C code cannot be guaranteed, due to the limitations of decompilation [39]. We also substantiate this point in Section 6. Instead of conducting the source code analysis on decompiled C code, XMATCH targets on the lifted binary code which is more accurate than decompiled C code. Furthermore, it is more explainable, since it can locate potentially vulnerable code logics in the function. This is substantiated in Section 6.

9. CONCLUSION

In this paper, we adopted feature representation *conditional formulas* to conduct the cross-architecture code search. The conditional formula explicitly captured two cardinal factors of a bug: 1) erroneous data dependencies and 2) missing or invalid condition checks. To better facilitate human bug verification, we formulated conditional formulas matching as a linear assignment problem and leverage integer programming techniques to correlate the statements in two binary programs in an optimal fashion. We had implemented a prototype, XMATCH, and evaluated it using the well-known software `OpenSSL` and `BusyBox`. Experimental results had shown that XMATCH outperforms existing bug search techniques. At the same time, it also provided evidence of detected vulnerabilities, which can then be easily examined via human inspection.

Acknowledgment

We would like to thank anonymous reviewers for their feedback. This research was supported in part by National Science Foundation Grant #1664315, Air Force Research Lab Grant #FA8750-15-2-0106, and DARPA Grant #FA8750-16-C-0044.. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

10. REFERENCES

[1] The LLVM Compiler Infrastructure. http://llvm.org/.

[2] The z3 theorem prover. https://z3.codeplex.com/, 2010.

[3] Dd-wrt firmware image r21676. ftp://ftp.dd-wrt.com/others/eko/BrainSlayer-V24-preSP2/2013/05-27-2013-r21676/senao-eoc5610/linux.bin(lastvisit: 2016-1-20), 2013.

[4] Retargetable decompiler. https://retdec.com, 2013.

[5] AVGERINOS, T., CHA, S. K., REBERT, A., SCHWARTZ, E. J., WOO, M., AND BRUMLEY, D. Automatic exploit generation. *Communications of the ACM 57*, 2 (2014), 74–84.

[6] BALAKRISHNAN, G., GRUIAN, R., REPS, T., AND TEITELBAUM, T. Codesurfer/x86: A platform for analyzing x86 executables. In *Compiler Construction*, Lecture Notes in Computer Science. 2005.

[7] BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *Compiler Construction* (2004).

[8] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. Bap: a binary analysis platform. In *Computer aided verification* (2011), Springer, pp. 463–469.

[9] Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy (Oakland)* (2006).

[10] Caballero, J., Johnson, N. M., McCamant, S., and Song, D. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium* (San Diego, CA, Feb. 2010).

[11] Cha, S. K., Woo, M., and Brumley, D. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)* (2015).

[12] Chen, D. D., Egele, M., Woo, M., and Brumley, D. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS* (2016).

[13] David, Y., and Yahav, E. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'14)* (2014), ACM.

[14] DINABURG, A., and RUEF, A. Mcsema: Static translation of x86 instructions to llvm. In *ReCon* (2014).

[15] Dolan-Gavitt, B., Leek, T., Hodosh, J., and Lee, W. Tappan zee (north) bridge: mining memory accesses for introspection. In *CCS* (2013).

[16] Dullien, T., and Porst, S. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest* (2009).

[17] Egele, M., Woo, M., Chapman, P., and Brumley, D. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security* (2014).

[18] ElWazeer, K., Anand, K., Kotha, A., Smithson, M., and Barua, R. Scalable variable and data type detection in a binary rewriter. In *ACM SIGPLAN Notices* (2013).

[19] Eschweiler, S., Yakdan, K., and Gerhards-Padilla, E. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS* (2016).

[20] Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., and Yin, H. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016).

[21] Gao, D., Reiter, M. K., and Song, D. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*. Springer, 2008, pp. 238–255.

[22] Geoffrion, A. M. *Lagrangean relaxation for integer programming*. Springer, 1974.

[23] Ireland, A., and Stark, J. On the automatic discovery of loop invariants. In *NASA Conference Publication* (1997).

[24] Jang, J. *Scaling Software Security Analysis to Millions of Malicious Programs and Billions of Lines of Code*. PhD thesis, CARNEGIE MELLON UNIVERSITY, 2013.

[25] Jang, J., Agrawal, A., and Brumley, D. Redebug: finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy (Oakland)* (2012).

[26] Jhala, R., and Majumdar, R. Path slicing. In *ACM SIGPLAN Notices* (2005).

[27] Kamiya, T., Kusumoto, S., and Inoue, K. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering 28*, 7 (2002), 654–670.

[28] Karim, M. E., Walenstein, A., Lakhotia, A., and Parida, L. Malware phylogeny generation using permutations of code. *Journal in Computer Virology 1*, 1-2 (2005), 13–23.

[29] Khoo, W. M., Mycroft, A., and Anderson, R. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (2013), IEEE Press.

[30] Kuhn, H. W. The hungarian method for the assignment problem. In *50 Years of Integer Programming 1958-2008*. 2010, pp. 29–47.

[31] Lee, J., Avgerinos, T., and Brumley, D. Tie: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium* (Feb. 2011).

[32] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI* (2004), vol. 4, pp. 289–302.

[33] Ming, J., Pan, M., and Gao, D. ibinhunt: binary hunting with inter-procedural control flow. In *Information Security and Cryptology*. Springer, 2012, pp. 92–109.

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

[35] Pewny, J., Garmany, B., Gawlik, R., Rossow, C., and Holz, T. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy (Oakland'15)* (2015), IEEE.

[36] Pewny, J., Schuster, F., Bernhard, L., Holz, T., and Rossow, C. Leveraging semantic signatures for bug search in binary programs. In *ACSAC* (2014).

[37] Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., and Brumley, D. Optimizing seed selection for fuzzing. In *USENIX Security* (2014).

[38] Riesen, K., Neuhaus, M., and Bunke, H. Bipartite graph matching for computing the edit distance of graphs. In *Graph-Based Representations in Pattern Recognition*. 2007, pp. 1–12.

[39] Schwartz, E. J., Lee, J., Woo, M., and Brumley, D. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security* (2013).

[40] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS* (2015).

[41] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (Hyderabad, India, Dec. 2008).

[42] Stephens, N., Grosen, J., Salls, C., Dutcher, A., and Wang, R. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS* (2016).

[43] Taha, H. A. *Integer programming: theory, applications, and computations*. Academic Press, 2014.

APPENDIX

A. THE DEMOS FOR MORE VULNERABILITIES

A.1 CVE-2014-3567

This vulnerability is caused by `tls_decrypt_ticket` function. Memory leakage in this function allows remote attackers to cause a denial of service via a crafted session ticket that triggers an integrity-check failure. This vulnerability incorrectly handles the session ticket cleanup by not adding the proper cleanup function. We can still capture its vulnerability in terms of conditional formulas. Unlike Heartbleed, this vulnerability is not about missing conditional checking on a specific statement, but rather about missing a specific conditional formula. In this case, as shown in Figure 15, XMATCH locates the extra conditional formula in the MIPS version that cannot be matched by any existing conditional formulas in x86. By comparing the two matched conditional formulas for both x86 and MIPS, we see that the bolded part indicates that an extra function is added under a certain condition. This is a clear explanation of why their function similarity is 0.90. Their difference also provides strong evidence that the target function is patched.

A.2 CVE-2015-0292

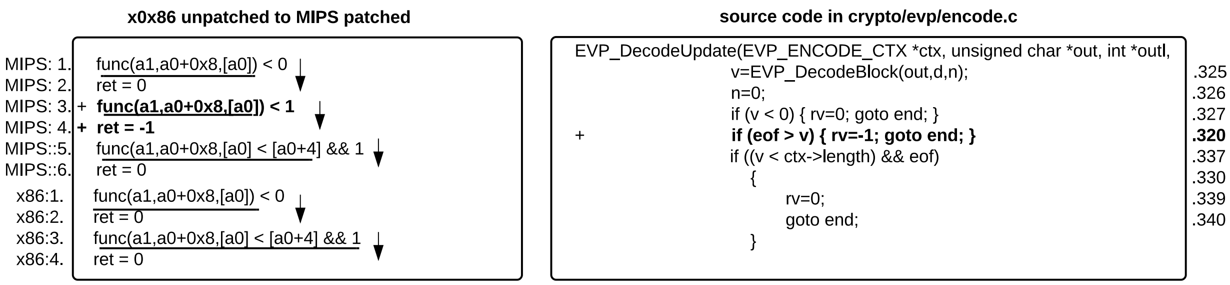
This shows the integer overflow in `EVP_DecodeUpdate` which allows remote attackers to cause a denial of service. The vulnerability is missing a specific conditional formula for boundary checking. XMATCH locates this missing conditional formula by comparing the conditional formulas of two matched functions. Its condition and statements clearly explain its semantics in in Figure 13. An expert can directly investigate this specific program logic and diagnose that the MIPS version is patched.

A.3 CVE-2015-1789

This allows remote attackers to cause a denial of service via a crafted length field in `ASN1_TIME` data. `X509_cmp_time` is responsible for this vulnerability. Although the patch in source code level spans several lines as shown in Figure 14, it is only represented as two lines of conditional formulas. This is because that most of the patch source code only renames names of the original variables, such as variables from line:1809 to line:1811. The real patch in the source code is on line:1820. This explains why the conditional formulas for this patch are so small.

A.4 CVE-2014-3508

This vulnerability is related to the function `c2i_ASN1_OBJECT`. By comparing it against the MIPS version, an expert can clearly understand that the MIPS version adds conditional formulas for variable checking in Figure 16.



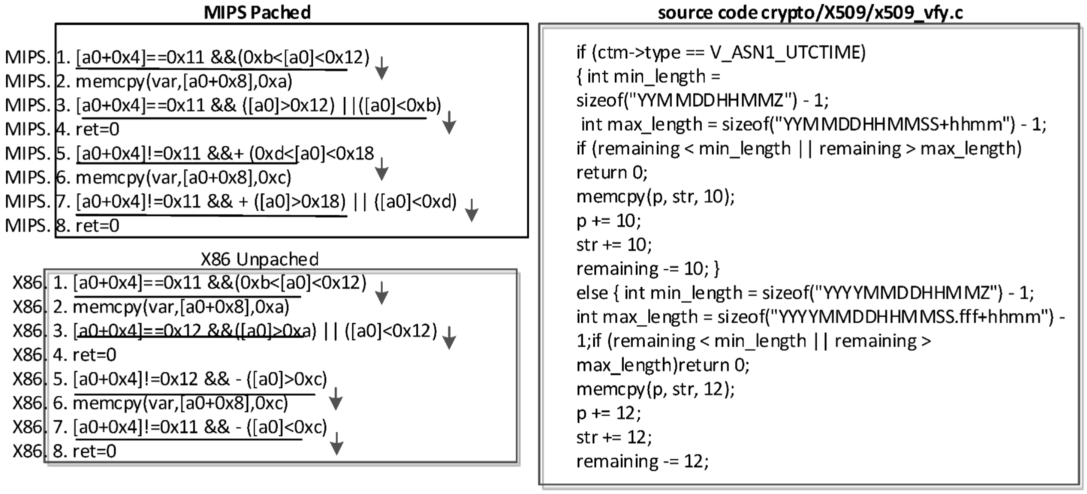
x0x86 unpatched to MIPS patched

```
MIPS: 1.    func(a1,a0+0x8,[a0]) < 0
MIPS: 2.    ret = 0
MIPS: 3. +  func(a1,a0+0x8,[a0]) < 1
MIPS: 4. +  ret = -1
MIPS::5.    func(a1,a0+0x8,[a0] < [a0+4] && 1
MIPS::6.    ret = 0
 x86:1.     func(a1,a0+0x8,[a0]) < 0
 x86:2.     ret = 0
 x86:3.     func(a1,a0+0x8,[a0] < [a0+4] && 1
 x86:4.     ret = 0
```

source code in crypto/evp/encode.c

```
EVP_DecodeUpdate(EVP_ENCODE_CTX *ctx, unsigned char *out, int *outl,
                         v=EVP_DecodeBlock(out,d,n);                      .325
                         n=0;                                             .326
                         if (v < 0) { rv=0; goto end; }                   .327
+                        if (eof > v) { rv=-1; goto end; }                .320
                         if ((v < ctx->length) && eof)                    .337
                           {                                              .330
                                rv=0;                                     .339
                                goto end;                                 .340
                           }
```

Figure 13: The explainability demo for `EVP_Decode_Update` vulnerability



MIPS Pached

```
MIPS. 1. [a0+0x4]==0x11 &&(0xb<[a0]<0x12)
MIPS. 2. memcpy(var,[a0+0x8],0xa)
MIPS. 3. [a0+0x4]==0x11 && ([a0]>0x12) ||([a0]<0xb)
MIPS. 4. ret=0
MIPS. 5. [a0+0x4]!=0x11 &&+ (0xd<[a0]<0x18
MIPS. 6. memcpy(var,[a0+0x8],0xc)
MIPS. 7. [a0+0x4]!=0x11 && + ([a0]>0x18) || ([a0]<0xd)
MIPS. 8. ret=0
```

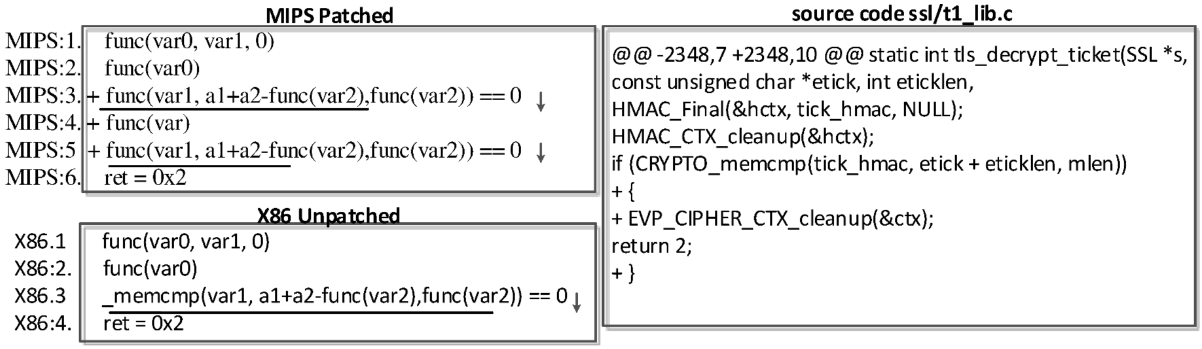
X86 Unpached

```
X86. 1. [a0+0x4]==0x11 &&(0xb<[a0]<0x12)
X86. 2. memcpy(var,[a0+0x8],0xa)
X86. 3. [a0+0x4]==0x12 &&([a0]>0xa) || ([a0]<0x12)
X86. 4. ret=0
X86. 5. [a0+0x4]!=0x12 && - ([a0]>0xc)
X86. 6. memcpy(var,[a0+0x8],0xc)
X86. 7. [a0+0x4]!=0x11 && - ([a0]<0xc)
X86. 8. ret=0
```

source code crypto/X509/x509_vfy.c

```
if (ctm->type == V_ASN1_UTCTIME)
{ int min_length =
sizeof("YYMMDDHHMMZ") - 1;
 int max_length = sizeof("YYMMDDHHMMSS+hhmm") - 1;
if (remaining < min_length || remaining > max_length)
return 0;
memcpy(p, str, 10);
p += 10;
str += 10;
remaining -= 10; }
else { int min_length = sizeof("YYYYMMDDHHMMZ") - 1;
int max_length = sizeof("YYYYMMDDHHMMSS.fff+hhmm") -
1;if (remaining < min_length || remaining >
max_length)return 0;
memcpy(p, str, 12);
p += 12;
str += 12;
remaining -= 12;
```

Figure 14: The explainability demo for `X509_cmp_time` vulnerability



MIPS Patched

```
MIPS:1.   func(var0, var1, 0)
MIPS:2.   func(var0)
MIPS:3. + func(var1, a1+a2-func(var2),func(var2)) == 0
MIPS:4. + func(var)
MIPS:5  + func(var1, a1+a2-func(var2),func(var2)) == 0
MIPS:6.   ret = 0x2
```

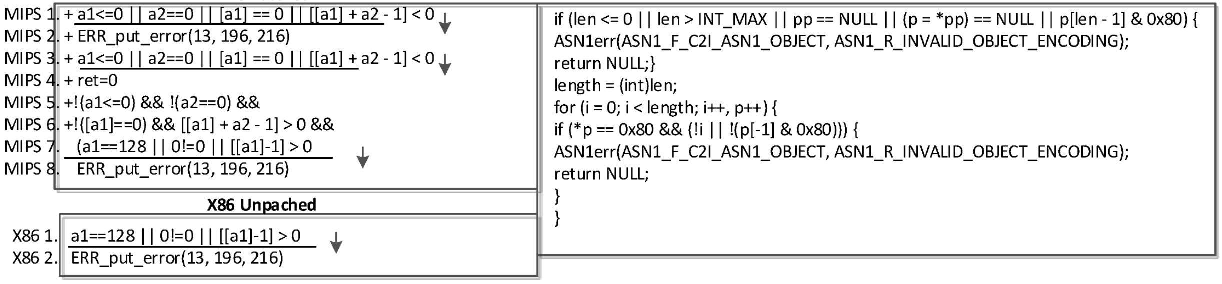
X86 Unpatched

```
X86.1    func(var0, var1, 0)
X86:2.   func(var0)
X86.3    _memcmp(var1, a1+a2-func(var2),func(var2)) == 0
X86:4.   ret = 0x2
```

source code ssl/t1_lib.c

```
@@ -2348,7 +2348,10 @@ static int tls_decrypt_ticket(SSL *s,
const unsigned char *etick, int eticklen,
HMAC_Final(&hctx, tick_hmac, NULL);
HMAC_CTX_cleanup(&hctx);
if (CRYPTO_memcmp(tick_hmac, etick + eticklen, mlen))
+ {
+ EVP_CIPHER_CTX_cleanup(&ctx);
return 2;
+ }
```

Figure 15: The explainability demo for `tls_decrypt_ticket` vulnerability



MIPS Patched

```
MIPS 1. + a1<=0 || a2==0 || [a1] == 0 || [[a1] + a2 - 1] < 0
MIPS 2. + ERR_put_error(13, 196, 216)
MIPS 3. + a1<=0 || a2==0 || [a1] == 0 || [[a1] + a2 - 1] < 0
MIPS 4. + ret=0
MIPS 5. +!(a1<=0) && !(a2==0) &&
MIPS 6. +!([a1]==0) && [[a1] + a2 - 1] > 0 &&
MIPS 7.   (a1==128 || 0!=0 || [[a1]-1] > 0
MIPS 8.   ERR_put_error(13, 196, 216)
```

X86 Unpached

```
X86 1.  a1==128 || 0!=0 || [[a1]-1] > 0
X86 2.  ERR_put_error(13, 196, 216)
```

source code crypto/asn1/a_object.c

```
if (len <= 0 || len > INT_MAX || pp == NULL || (p = *pp) == NULL || p[len - 1] & 0x80) {
ASN1err(ASN1_F_C2I_ASN1_OBJECT, ASN1_R_INVALID_OBJECT_ENCODING);
return NULL;}
length = (int)len;
for (i = 0; i < length; i++, p++) {
if (*p == 0x80 && (!i || !(p[-1] & 0x80))) {
ASN1err(ASN1_F_C2I_ASN1_OBJECT, ASN1_R_INVALID_OBJECT_ENCODING);
return NULL;
}
}
```

Figure 16: The explainability demo for `c2i_asn1_object` vulnerability