The Pennsylvania State University

The Graduate School

College of Engineering

DETECTING NON-CONSTANT TIME CODE IN CRYPTOGRAPHY LIBRAIES USING A STATIC INFOMRATION FLOW ANALYSIS

A Thesis in

Computer Science and Engineering

by

Adam Mohammed

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2018

The thesis of Adam Mohammed was reviewed and approved* by the following:

Danfeng Zhang
Professor of Computer Science
Thesis Adviser

Gang Tan
Professor of Computer Science

Abstract

Identifying side-channels for crypto-systems is often a manual process. Addressing the cause of the side-channel is only possible one the flaw is identified. Using information-flow analysis it is possible to identify IR level side-channels. For example, a branch dependent on secret data may cause a timing side channel if the operations for each path are distinguishable by execution time. An information flow analysis can track the flow of sensitive information through a program. The issue is that the number of positives can be very high, and unequally weighted in the threat that may be posed. A static information-flow analysis can be used and the precision adjusted to improve the precision of the analysis.

Improving the precision, means decreasing the overall number of positives while maintaining all of the higher-risk positives. In this work, an information flow analysis and memory abstraction is used in conjunciton to detect secret-dependent branches. Adding field-sensitivity in some cases improves the precision of the analysis. In some cases, field-sensitivity is not able to reduce the number of positives, so a white-list can be used to ignore positives that have been ruled out.

The baseline analysis combined with the field-sensitivity and whitelist were used in a case study exploring the effectiveness of each feature on real-world crypto-systems. The modular exponentiation function was analyzed as it is used in public-key cryptography and side-channel attacks have been identified in the code providing this functionality. The results show that improving the precision can be helpful to reduce the number of positives.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Side-channel vulnerabilities can be found in modern cryptographic systems that are widely adopted. These widely adopted systems are used to provide confidentiality and integrity of data communicated between parties. Attackers can exploit these side-channel vulnerabilities to compromise the assumptions of these safe communication methods. Side-channels exist in many forms and are often the result of implementation details which are not available in theoretical proposals of a protocol.

Many methods have emerged as a response to automate the process of identifying side-channel vulnerabilities. Currently as vulnerabilities are identified, the measures taken to mitigate these vulnerabilities are highly specific to the problem. It is important to be able to analyze software to identify possible side channels, and mitigate them as necessary. The problem becomes identifying the vulnerabilities. Static analyses exist to be able to identify these vulnerabilities.

Static analyses can struggle with the number of false positives being fairly high. The precision is often sacrificed to decrease the runtime of an analysis. Practical applications of a static analysis would optimally have the least amount of false positives while remaining sound. Developers and engineers who design and implement these libraries can then more easily focus on the implementations most likely to be vulnerable.

There are many types of static analysis targeted at identifying side-channels. There have also been works which analyze cryptographic protocols using proof-engineering techniques. The issue with these approaches is in terms of scalability and accurate representation of system state. Static analysis tools have the ability to work directly with the binary so the state is closer to the actual state during execution of the software.

This work is based on a static analysis which tracked information-flow in order to find variables computed from secret values as well as find secret-dependent branches. The analysis uses a memory abstraction which allows the static analysis to handle dynamically allocated memory. The information-flow analysis is heavily reliant on the memory abstraction to track the information flow of practical programs. The precision of the information-flow analysis is influenced by how it integrates with the memory abstraction.

This work looks at the effects of adding extra features to the baseline information-flow analysis to increase precision. The positives that are eliminated should not be any high-risk positives and the total number should be fewer than the baseline. In the case study, the analysis is run on modular exponentiation source code used in popular crypto-systems, with additional features. There were two features added to the baseline analysis to try and achieve this goal, i) field-sensitivity, ii) white-listing. Lastly, the analysis has to be conservative when source code for a given function call is not provided, so the analysis was tested with and without the full library source code.

The field sensitivity is improved by relying on more information provided by the memory abstraction. The white-list is aimed at removing results that may actually be leaking information but are considered acceptable. The white-list can also be used to ignore results which cause other erroneous positives.

A case-study is performed by analyzing the modular exponentiation source code provided for Libgcrypt, OpenSSL, mbedTLS, and BearSSL. The results help identify what features are necessary to improve the precision of an information-flow analysis.

Chapter 2

Background

2.1 Information Flow Analysis

Information flow analysis tracks interactions of information throughout a program. A simple application of this analysis is used in LOMAC to restrict access to data of various integrity levels. LOMAC maintains a concept of information integrity for each object, and restricts information from flowing from low integrity objects to high integrity objects. If a program source is inspected, and data is given a level of integrity or confidentiality then by following the flow of information in the program, it is possible to track which data have been computed from confidential or high integrity information. In cryptographic systems, the encryption keys are confidential, if other data within the program is calculated from the encryption key, there is a flow of information from the confidential data to the computed data [**?**]. Compilers also use information flow for optimization purposes. By identifying what information is used and when, alterations to a program can be made to improve cache locality, hide memory latency and improve performance in other manners. For program level security, a compiler could be able to check the level of integrity for any variable through the use of information flow. By tracking confidential or sensitive pieces of information, any variable or decision which is computed from a confidential data has a flow from the confidential data to the variable or branch. An attacker may be able to exploit parts of a program which are dependent on some confidential data, and with sufficient flow, can possibly reconstruct the confidential data. An example is given in Algorithm 2.1 to illustrate this concern.
In this simple example k is confidential data, through information flow we

```
k = sensitive information
d.x = k + 1
d.y = 0
a = −1
if d.y == k/2 then
    a = 4
end if
```

Figure 2.1: Simple Information Flow

can see that the value of $d.x$ is directly computed from k. The value $d.y$ however is not confidential due to its value having no reliance on the confidential data. The flow between k and $d.x$ is called explicit flow, since the value of $d.x$ is computed from the confidential data. As the analysis continues, any value which is computed from $d.x$ will also be treated as confidential data. There is also implicit flow which can be seen in the first branch on $d.y$ where it is compared with some value computed from k. In this case $d.y$ is still considered non-confidential data, since the value has no relation to k, but there is a flow from k to a indirectly. Although, a is not computed from k, after the branch executes, the value of a may or may not have changed. If it does change, the value of k has been determined. If it does not change, the value of k is not determined, but a possible value has been ruled out. This indirect flow from k to a is what is called implicit flow. Tracking the flow of information between variables can be done by creating a set of constraints for each instruction to represent the direction of flow. Creating these constraints with mutable pointers, requires more information than is provided from IR code from the compilation process. Information flow analyses have been capable to find even hardware based vulnerabilities from analyzing the source. Information flow has been used to detect leaking of confidential data through the source and also leaking data through hardware side channels such as through the cache [**?**].

2.2 Points-to Analysis

A Points-to analysis is a reference tracking analysis to identify the targets of pointers in a program. Information flow alone is not enough once pointers are involved. For example, in Algorithm 2.1, it must be possible to identify from the source the location of d, and subsequently the location

```
1 for (i = 1; i < bits; i++) {
2     if (!BN_sqr(v, v, ctx))
3         goto err;
4     if (BN_is_bit_set(p, i)) {
5         if (!BN_mul(rr, rr, v, ctx))
6             goto err;
7     }
8 }
```

Figure 2.2: Square and Multiply Timing Channel

of x. The points-to analysis provides this information so that the proper location of x and y can be known even if the value of d changes, meaning it points to another location in memory. In addition to providing spatial information, the points-to analysis used in this research provides type and data layout information. The analysis accomplishes tracking by maintaining a internal graph representation of memory throughout the program. The graph is made up of nodes which are units of memory that are able to be the target of a pointer. Data structures identified in program source have one node for an instance of the data structure, where all fields pertaining to that data structure are represented by that node. The edges in the graph are references, so it is possible to identify where the target of a pointer resides. Additionally, this analysis provides type information along with offsets of the type from the base pointer. In the baseline tainted flow analysis, the granularity of the results was achieved by creating constraints of information flow between nodes from the points-to graph. This however, as will be discussed, leads to an over-approximation of the resulting tainted values. The points-to analysis used also has a method of determining the validity of the type information.

2.3 Timing Channels in Crypto-systems

A timing channel is a side channel in which an attacker uses execution time to learn information about sensitive data. In some implementations of sliding window exponentiation, the sequences of squares and multiplies could be measured due to differences in each methods execution time. This attack though timing based can be observed in source code in OpenSSL's previous implementation of sliding window exponentiation, shown in Algorithm 2.2.

The for-loop here iterates through the number of bits in the confidential

power p, each time through squaring `v` until a set bit is encountered in `p`. Once a set bit is encountered, an additional multiply step is executed before proceeding to the next loop iteration. This lends itself to a timing attack due to multiplication more expensive to compute than a square. The timing channel exists because the number of operations executed during one iteration changes based on the value of the confidential data. The result is if one can determine the locations of the multiplies and the length of time expected between each multiplication, the key can be rebuilt as has been done by Bernstein et al [**?**]. There are other forms of timing channels which may or may not be viewable in source code, such as disk access timing or network timing attacks.

Chapter 3

Methodology

In this work, improvements were made to the precision of the baseline taint analysis. The baseline represented each structure or array as one entity, which led to imprecision when tainted data flowed to only some parts of an entity which had multiple fields or elements. Simply, the improved analysis generates multiple elements to sufficiently represent different data structures and constrains the appropriate ones based on the operation. Improvements in precision also increased the accuracy of the analysis by reducing the number of false positives from the final results. The analysis is comprised of following 3 components: Flow capture, constraint generation, and solution.

3.1 Information Flow Capture

Given a program in the LLVM IR representation, flow of information is tracked by identifying the operands. Each operand can be considered as a source or a sink, though this alone is not enough information when considering the effects of pointers. Since pointers can point to different data and the data to which they point can be changed, the points-to analysis is used. The points-to analysis is used to keep track of which memory locations are associated with the LLVM IR values identified as sources and sinks.

For each LLVM value, there are 3 elements which can be constrained. There are 3 transformation functions which returns the element to be constrained for that value. For any operations, zero or more of the three functions (V, D, R) may be used in generating a constraint for the instruction.

- $V(x)$ is the constraint element associated with the IR value x.
- $D(x, offset)$ is the constraint element associated with the memory represented by IR value x and the $offset$.
- $R(x)$ is the set of all reachable memory locations that are represented by IR value x.

The baseline analysis, also used these same transformations, but the addition of field sensitivity changed the behavior of the functions D and R. The memory nodes of these were only able to be constrained as a single entity. Since a whole data structure was referred to as a single element, tracking the flow to individual fields was not available. The points-to analysis provides a type map, which can be used to identify the field type and offset within the data structure. Using this information, the D and R functions consumed an extra operand to return the element to be constrained for a particular field.
For most IR instructions, the explicit flow is constrained by adding the constraint $V(source) \rightarrow V(sink)$. There are a subset of LLVM IR memory operations that require interaction with the points-to analysis. The store, load and GetElementPtr instructions are the most important for addressing the issue of field sensitivity.

3.1.1 Instruction Flow Rules

Figure 3.1 shows the flow rules for the memory operations. For stack allocations, the points-to analysis just creates a node with no type information. The new node is marked so that it is identifiable as a stack memory node. The memory flows will be constrained once a load or store occurs.
The flow rules for the GEP instruction do not include the offset, the offset is accounted for when the load or store instruction flows are generated. For all instructions apart from the loads and the stores, the information flows are the operands values to the result of the operand. Constraining loads and stores properly requires a bit more context to be able to create a field-sensitive analysis.

3.1.2 Constraint Generation

For operations which are not modifying memory, generating the constraints are done through the form $V(inputs) \rightarrow V(outputs)$ if there are any

```
<result> = alloca <type> [, <ty> <NumElements>] [, align <
    alignment>]
```
Explicit: $\rightarrow V(\text{result})$
Implicit: $V(PC) \rightarrow V(\text{result})$

```
<pointer> = getelementptr inbounds <ty>, <ty>* <ptrval>{, [
    inrange] <ty> <idx>}*
```
Explicit: $V(\text{ptrval}) \rightarrow V(\text{pointer})$
Implicit: $V(PC) \cup V(\text{ptrval}) \rightarrow V(\text{pointer})$

```
store <ty> <value>, <ty>* <pointer>[, align <alignment>]
```
Explicit: $V(\text{value}) \rightarrow D(\text{pointer})$
Implicit: $V(PC) \cup V(\text{pointer}) \rightarrow D(\text{pointer})$

```
<result> = load <ty>, <ty>* <pointer>[, align <alignment>]
```
Explicit: $V(\text{pointer}) \cup D(\text{pointer}, \text{offset}) \rightarrow V(\text{result})$
Implicit: $V(PC) \cup V(\text{pointer}) \cup D(\text{pointer}, \text{offset}) \rightarrow V(\text{result})$

Figure 3.1: LLVM Memory IR Flow Rules

relevant sources/sinks. Loads and store instructions require additional steps to include the points-to analysis. The points-to analysis represents memory as nodes in a graph. Structures and arrays are represented as a single node with as many types and offsets as necessary. Figure 3.2 shows how a store would be constrained using additional information from the pointer operand and the type information from the points-to analysis.

```
1 typedef struct {
2   int rd_key;
3   int rounds;
4 } AES_KEY;
5
6 AES_KEY pk;
7 pk.rounds = 3;
```

```
1   %pk = alloca %struct.AES_KEY, align 4
2   %rounds = getelementptr inbounds %struct.AES_KEY, %struct.
        AES_KEY* %pk, i32 0, i32 1
3   store i32 3, i32* %rounds, align 4
```

Figure 3.2: Store Constraint Example

In Fig. 3.2 The *AES_KEY* structure has two fields. The GEP instruction is used to calculate the address of the fields when operating on the

structure. The inputs are the base pointer and zero or more offsets. The store instruction is constrained using the pointer and the offset operands, which are provided by the GEP instruction that flows into the store instruction. In this example, the offset to be used is the last operand. The last operand is not a byte offset from the base pointer, its an index. Using the points-to analysis, the index is converted into a byte offset. Bytes are used instead of the index to account for a types width. The type width matters in the cases where types are overlapping.
The rest of this section shows how the analysis handles when the necessary information is not available to create a proper field sensitive constraint.

3.1.3 Factors affecting sensitivity

Collapsed Memory Nodes

The points-to analysis is type-safe, marking nodes as collapsed when the type information is inconsistent. If a node is collapsed, only one constraint element is created for that node. Multiple constraint elements for a single memory node are only created when the points-to analysis has type information available for the node.

Incomplete Type Information

The points-to analysis used only provides type information for the values used in the analyzed source. The field-sensitive analysis should also be conservative when analyzing unknown source. For this, the analysis relies on the data layout of structures which is the same information the points-to analysis uses to build its type information. The layout information helps in the case where a tainted field is not used within the provided source but its data structure is passed as an argument to an unknown function. This provided the possibility for the tainted field to leak information.

Calls to Unknown Functions

For functions which the source code is not provided, the constraint rules define how to treat values and their flow. The rules used, allow for flow between all pointer types, and also flow of information within the pointer itself. That is, a structure may only have a single tainted field, but after passing through an unknown source, all fields are tainted due to possible

flow between fields. Likewise, flow from one pointer to another ensures that any tainted value propagates in the constraints to other parameters.

$$V_{args} = \bigcup_{i=0}^{N} V(arg_i)$$

$$R_{args} = \bigcup_{i=0}^{N} \{R(arg_i) : type(arg_i) = pointer\}$$

Explicit: $V_{args} \cup R_{args} \rightarrow V(retval) \cup R_{args}$
Implicit:$V(PC) \cup V(functionptr) \cup V_{args} \cup R_{args} \rightarrow V(retval) \cup R_{args}$

3.2 Solving Set of Constraints

Given the set of constraints for the program being analyzed, the least solution where all the sinks are greater than or equal their sources is found, otherwise known as a least solution. From the solution each sink will have a value is greater than or equal to that of the initial confidential data, then that value is identified as being confidential as well. In the case of crypto-systems, it can be helpful to find all the values which may have been computed from confidential data. These values are possible locations for leakage of confidential information, presenting a possible vulnerability. Consider figure 2.1 for example of how the analysis functions. First the flow of information must be identified. For simplicity, this example will just consider the source code, not the instructions. The more detailed analysis will be done in the implementation section. In figure 2.1, k is just a constant set to some confidential value, no information flows there. Next, $d.x$ is calculated by adding 1 to the value of k, so there is flow from k to $d.x$. With $d.y$ and a, in lines 3 and 4 respectively, they are set with constants, so no flow happens there. Then, a branch is encountered where the value of a is modified based on the values of $d.y$ and k. Let the branch condition be b, there will be flow from $d.y$ and k to b. Lastly due to the branch, there is an implicit flow from $d.y$ to a and k to a.
For explicit flow the set of constraints is as follows.

$$k \leq d.x$$

$$d.y \leq b$$

$$k \leq b$$

In this case it is easy to see that $d.x$ should be at least as confidential as k since it is directly computed from k. Similarly, the branch condition b is confidential since it must be at least as confidential as k. The value $d.y$ is not confidential, but is irrelevant because confidential information is used to determine the outcome of the branch.
If implicit constraints are to be considered then the set is as follows.

$$k \leq d.x$$

$$d.y \leq b$$

$$k \leq b$$

$$b \leq a$$

In this case the all the constraints are the same apart from the addition of the last constraint. There is another constraint between the branch condition and a, because after the branch, the value of a is dependent on the outcome of the branch. If an attacker had a method of inspecting the value of a, they would learn information about b which is derived from k.
The baseline analysis generates a set of constraints similar to the examples above, but with some imprecision. During the stage of generating constraints from the information flow, the points-to analysis is leveraged to figure out what is being pointed to by the variable d to subsequently find the correct instance of x. The points-to analysis will represent the data structure pointed to by d with one node in the graph, and the constraint generated refers only to that node, instead of to the specific field in the node. The effect on the constraints is as follows for explicit flow.

$$k \leq d$$

$$d \leq b$$

$$k \leq b$$

The result is that $d.x$ and $d.y$ are indistinguishable from each other and the results will report both $d.x$ and $d.y$ as confidential even though manual scanning of the source shows that $d.y$ is never calculated from confidential data. This is a common problem when analyzing sources. Take this implementation of AES, where the data structure used is comprised of both confidential and public data. The *AES_KEY* struct has the confidential *rd_key* and a second field rounds that is considered to be public. Figure 3.3 shows where the inaccuracy can be found in actual source code.

So with the improved analysis, the constraints are generated such that compound data structures with multiple nodes are able to be constrained separately. Once the set of constraints for a programs source code is created, the least solution is found and the results show the line number where the branch occurs. This list of results shows the branches which are tainted and untrusted.

```
1  struct AES_KEY_t {
2    unsigned * rd_key;
3    int rounds;
4  };
5
6  typedef AES_KEY_t AES_KEY;
7
8  void AES_enc(char * in, char * out, AES_KEY) {
9    unsigned * rk = key->rd_key;
10   if (key->rounds > 10) {
11     ...;
12   }
13 }
```

Figure 3.3: Public and private data in structure

3.3 Result Classification

The analysis provides a list of source code lines which are vulnerable. Based on the contents of the input files, a branch is flagged as vulnerable if the condition depends on data which has been marked as tainted or untrusted. The reported lines can then be reviewed and sorted to rank them in order of severity. Classification of the vulnerable branches consists of three stages, the first removes error-handling results, the second sorts the remaining results into either a high or low-risk set, and the final sorts the high-risk results depending on the surrounding source context.

Stage 1 is a filter which removes the error handling and input validation results. For normal operation of a program, these branches check sensitive data but allow an early exit if the branch condition evaluates to true. In figure 3.4, the variable `N` is tainted, so the branch is vulnerable due to the condition depending on `N` and the data held within `N`. The result of this branch is not helpful in the case of an attacker attempting to learn the value of `N`. As long as the input is valid, the condition of this branch will

always evaluate to false. If the branch leads to exit code, it is a candidate for removal in this stage.

```
1  if( mbedtls_mpi_cmp_int( N, 0 ) <= 0 || ( N->p[0] & 1 ) == 0
       )
2      return( MBEDTLS_ERR_MPI_BAD_INPUT_DATA );
```

Figure 3.4: Validation Source Code

Stage 2 operates on the remaining results to sort them into high and low-risk categories. A high-risk branch will be due to one of the operands being either directly related to the sensitive data or derived from more than 1 bit of sensitive data. Low-risk branches, are branches which the operand is based on 1 bit of the key or the same operand could derived from some set of sensitive data values.

Figure 3.5 is a snippet of some of the modular exponentiation code from Libgcrypt 1.8.2. For instance, say the values pointed to by `ep` are marked as tainted at the start of the analysis, and the analysis reports that lines 10 and 11 are vulnerable. Line 10 is the macro defined in the lines 1-8. Line 10 is reported due to the macro having the `if` branch directly dependent on the data pointed to by `ep`. Since the branch is directly dependent on the tainted data that branch is considered high-risk. Line 11 however is a branch based on `esize`, the value begin modified within the `MPN_NORMALIZE` macro. The value of `esize` is not unique to the tainted data, meaning there exists a set of tainted values which may yield the same `esize`. If a value is derived from tainted data, as `esize` is, but the result is not unique to that instance of tainted data, it is considered low-risk.

If for any reason, the classification is unclear, such as a mutable type being passed to source code that is not analyzed, the data is treated as high risk. This is done in order to be conservative. Further categorization can be performed on the high-risk set of results in Stage 3. Additionally, if there are a large number of low-risk result, Stage 2 can be repeated, by appending the low-risk variables to the whitelist.

Stage 3 specifically looks a the context of the high-risk results from Stage 2. Each of the results will be classified as one of the following: a single branch, repeated branch, controllable branch or both (e.g. both controllable and repeated). Single branches are branches which are not within a loop. Repeated branches are branches that are part of a loop definition or within the body of a loop. Controllable branches are that which the branch result is both untrusted and tainted. The purpose of

```
#define MPN_NORMALIZE(d, n) \
do {                        \
  while( (n) > 0 ) {        \
    if( (d)[(n)-1] )        \
      break;                \
    (n)--;                  \
  }                         \
} while(0)

MPN_NORMALIZE(ep, esize);
if (esize * BITS_PER_MPI_LIMB > 512)
  W = 5;
```

Figure 3.5: High-Risk and Low-Risk Branches

sorting results this way is to be able to prioritize which vulnerable branches to examine first when attempting to make the application secure.

Chapter 4

Evaluation

4.1 Implementation

Achieving the improved precision of the analysis, is done primarily through handling LLVM's pointer instruction differently than other instructions. The pointer instruction in the IR is the GetElementPtr instruction. This instruction is used when computing an address from a base pointer. For arrays, for example it has the index of the element and the bounds of the array if they are known. For structures, the instruction contains the structure type which is being addressed and the index of the field in the structure that is being referenced. The baseline analysis lacked the ability to consider the index available in these instructions. The points-to analysis had type and offset information for the targets of the pointers. Improving the analysis was achieved by i) creating the appropriate number of constraint elements for each node in the points-to graph, ii) converting the field index from the GEP instruction to a byte offset, and iii) constraining the correct elements based on the GEP instruction and data type information.

4.1.1 Constraint Element Generation

When a pointer value is encountered for the first time, a node is created for it in the points-to graph, and a set of constraint elements are created for that data structure. The points-to analysis was left unmodified, so the focus will be on the semantics of how constraint elements were changed. In the baseline, only one constraint element represented any target of a pointer. In the improved analysis, type information from the points-to analysis was used to generate the appropriate number of constraint

elements. In a stack allocated array, one constraint element was created for each element in the array. For structures and classes, each field is located at some offset away from the base address of the structure and a constraint element was created for each field and mapped to associated byte offset. This change enables the precision of the baseline to be improved by correctly selecting the corresponding constraint element based on the IR. For any GEP instruction, constraint elements are generated using the type information from the points-to analysis when available. Each constraint element is created by iterating through the type information of the node from the points-to analysis. For each type there is an associated byte offset, using this offset and the size as reported by LLVM, a constraint element is created with the corresponding starting byte offset and the ending byte offset based on the width of the field. It is necessary to account for the width and start location of each field because there may be padding between fields of a structure. Padding between fields may be present in the type information because of an unused field in code. The points-to analysis will not have type information on unused fields so it is important to account for these gaps. Another reason to account for the padding is due to structures which are not aligned with each field following sequentially after the previous field. When the type information is not available one constraint element is created for the whole node so at worst the improved analysis will be equivalent to that of the baseline.

Figure 4.1 shows how the points-to analysis is leveraged to create constraint elements for the proper byte offsets and widths. Let `s` be the LLVM representation of the structure type referenced by the GEP instruction, and `node` be the node in the memory graph provided by the points-to analysis. The algorithm works by retrieving the graph in which the node resides and then retrieving the data layout as specified by the compiler for that graph. The data layout contains information like the field lengths of each type and the alignment within a data structure. The structure type itself `s` does not have the alignment and padding information, only the types of each field within the data structure. The data layout is used to retrieve the structure layout of the structure type from the GEP instruction. The structure layout is vital because it allows the index from the GEP instruction to be converted to a byte offset within the data structure. The data layout also provides the size of each type, so that the constraint element is created at the corrects starting and ending offset.

The elements created using this strategy enable the analysis to be more precise. GEP instructions can also be used to index arrays, so that is

```
1  const DataLayout &TD = node.getParentGraph()->getDataLayout
       ();
2  const StructLayout *SL = TD.getStructLayout(s);
3  int index = 0;
4  for(Type::subtype_iterator it = s->element_begin(); it != s
       ->element_end(); ++it, ++index){
5      unsigned start = SL->getElementOffset(index);
6      unsigned end = start + TD.getTypeStoreSize(*it);
7      std::string label = "[" + std::to_string(start) + "," +
           std::to_string(end) + "]";
8      const ConsElem & elem = kit->newVar(name+label);
9      locConstraintMap[&loc].insert(std::make_pair(start, &
           elem));
10 }
```

Figure 4.1: Creating constraint elements for each field in a type

handled by computing the number of elements that array may hold. For stack arrays that number is known, but for heap arrays the number is variable. For stack arrays each, since the number of elements is known, one constraint element is created per array element. For heap arrays, one constraint element is created for the entire array.
Similarly, if type information is unavailable then a conservative approach is used. One constraint element is created for the entire data structure. When the information is available, the analysis is able to be improved by constraining the correct constraint element for each instruction. When that information is unavailable, the improved analysis is unable to be more precise than the baseline analysis.

4.1.2 Constraining Operands

Each time an instruction where information flow exists, constraints are generated between the operands of that instruction. The idea of this is simple, given a set of constraint elements select the ones which are used in the operation and constrain them as necessary. The baseline analysis handled individual variables in a precise manner as long as the variable was not a structure or array. The baseline analysis had no way of identifying the offset from a GEP instruction, and even if it did, the mapping between a memory node to a constraint element was one-to-one. In the previous section, the mapping from memory node to constraint element was made to be one-to-many. As one node may represent multiple fields which together make the structure or array. It is now necessary to pick the correct

element from this one-to-many mapping when generating the constraint for each instruction. Again, this is done by analyzing the GEP instruction. The last operand of a GEP instruction is the index of the field within the data structure. Each constraint element for a memory node is created for a byte offset range, so it is necessary to correctly calculate the byte offset from the field index provided by the GEP instruction. This is very similar to the strategy used to generate the constraints at the correct byte locations. Given the structure type, field index and node from the points-to analysis, the offset is calculated as shown in figure 4.2. Given the node, the data layout and associated structure layout can be found and then the index can be converted to a byte offset.

```
1 unsigned findOffsetFromFieldIndex(StructType* type, unsigned
      fieldIdx, AbstractLoc* loc) {
2   DataLayout &TD = loc->getParentGraph()->getDataLayout();
3   StructLayout *SL = TD.getStructLayout(type);
4   return SL->getElementOffset(fieldIdx);
5 }
```

Figure 4.2: Calculating byte offset from index

If the GEP instruction is in reference to an array access, then the offset is just calculated using the size of the element type and the index. If no type information exists for the node, only a single constraint element would exist for the node, so that is the constraint which is selected to be constrained.

4.1.3 Missing Type Information

The type information used to generate the constraints came from the points-to analysis. The analysis is conservative in the type information it provides. This means that the type information given is limited to only fields that have been used in the source, and the type information is cleared if the analysis can not make definitively determine the types along with their offsets. The requirement to have a field used in the source means that often the type information from the structure is incomplete due to unused fields or fields which external functions may use. The points-to analysis clearing type information removes the ability to have a field sensitive constraint at all.

In the case of missing the unused data information, the solution is to utilize the information that the points-to analysis uses to build the type map. Given a structure type in LLVM, the list of fields and their types are

known. The points-to analysis relies on the data layout provided in the bitcode. Constraint elements are created by querying the data layout to identify the initial byte at which the type starts and the width of the type to find where that type should end. The start and end of the elements are recorded when creating the constraint elements for that type.

Building the type information always instead of on-demand is necessary when there are external function calls where the source is not linked. If a pointer which holds sensitive data is a parameter for that function, then it is possible that sensitive data may flow to other arguments or the return value. When the field-sensitivity was implemented, some of the higher-risk results had been removed. This happened because the pointer to sensitive data was no longer considered sensitive as it was when offset was unused. Before, it was enough to see if any of the arguments were sensitive, and create flows from the sensitive data to all sinks from that function. After the field sensitivity change, the pointer and its data are constrained separately, so when external calls happen, the analysis creates constraints from each of the fields which are reachable from that pointer in addition to the normal flows which are provided by the function call.

If there is no type information at all, then the improved and baseline analysis function the same way, there is one element that represents that type and that one element is constrained regardless of offset. The points-to analysis denotes unsafe type information by collapsing the node in the memory graph.

4.2 Benchmarks

The baseline analysis had many false positives and low-risk positives. Improving the analysis would mean reducing the positives to only results which are possible flaws. The benchmark results shown in table 4.1 show the effects of different features on the number of reported results. There are 2 features implemented which were looked at in isolation, the white-list and the field-sensitivity changes. Additionally, adding sources for functions which are not defined in the file being analyzed allows for a better understanding of information flow.

The baseline benchmark, evaluated the modular exponentiation code for each library. This benchmark was run without white-list or field sensitivity. The source provided was that only which was necessary to compile the modular exponentiation code. All multi-precision integer (mpi) arithmetic library components were omitted.

The field-sensitive (FS) test, is the same as the baseline benchmark but with field-sensitivity enabled. The results from this test differ depending on the library. For example, OpenSSL did not see any benefit from FS alone. This is due to openSSL doing much of the math in external functions, and each external function return value was checked in the modular exponentiation code. Since field-sensitivity requires source to work, it makes sense that openSSL did not see benefit in this test. The reasoning is the same for the result in mbedTLS. Libgcrypt however does manipulate the tainted data within the source analyzed so it did benefit from FS over the baseline benchmark.

The white-list (WL) test, is the same as the baseline benchmark but only using a white-list to remove results which are clearly false- or low-risk positives from the baselines results. The white-list allowed the libraries which did not benefit from field-sensitivity to see a reduction in positives. Libgcrypt saw an improvement as well but not nearly as much as the field-sensitive change.

The full source (SRC) test, is the same as the baseline but with additional source files provided. The additional sources provided were the remainder of the multi-precision integer or big number(bn) libraries. This meant that all mpi or bn math library calls were known functions. The important fact here is that the flows were more direct now between parameters in functions. Without field-sensitivity though all data-structures and arrays are treated as one element. This improved the number of results in all the libraries which did not benefit from field-sensitivity.

The last two tests were combining features to see if greater improvement was possible. The white-list and field-sensitivity test was chosen since the field-sensitivity is not able to function when external functions are called frequently, but white-listing can be used to compensate. The last test combined the 3 individual tests, using the baseline analysis with field-sensitivity, a white-list and the entire mpi/bn libraries.

The full combination of improvements over the baseline is where we get to see that adding additional source in combination with the field-sensitive changes improved the results in OpenSSL and mbedTLS the most. Libgcrypt however did see an increase in the number of positives, which is result based on the limitations of the points-to analysis used. The points-to analysis, makes use of heap-cloning to track distinct nodes which may be processed by a common source function. The analysis also merges nodes that are found to be in the same equivalence class. In the case of Libgcrypt, nodes that were considered distinct in the previous benchmarks, end up aliased to the same node in the full benchmark. This causes an

Library	Base	FS	WL	SRC	WL/FS	WL/FS/SRC	% Reduction
Libgcrypt 1.8.2	64	32	38	64	18	24	72.7%
mbedTLS 2.9.0	40	40	35	25	35	22	45.0%
BearSSL 0.5	1	1	1	1	1	1	0%
OpenSSL 1.1.0g							
Reciprocal	32	32	20	24	20	13	59.4%
Mont.	38	38	28	30	28	21	44.7%
Mont Const. Time	30	30	27	16	27	14	60.0%
Mont. Word	31	31	21	21	21	12	54.8%

Table 4.1: Number of Positives based on Features

inferior result to the WL/FS test.

4.3 Case Study

The baseline and improved analysis were compared using current software TLS and SSL libraries. The libraries analyzed were Libgcrypt, OpenSSL, mbed TLS, and BearSSL. The case study was done by comparing the modular exponentiation algorithms between each of the libraries. The Libgcrypt and OpenSSL libraries were chosen due to their wide-spread use. The mbed TLS library is built for embedded platforms, so this was chosen to search for shortcuts that may provide vulnerabilities. BearSSL is chosen because it claims to be a constant-time cryptographic library.

4.3.1 Experiments

The parameters used to test the libraries were set such that the data alone was set to be tainted, meaning that no other members which shared the same structure should be treated as sensitive data. The results of the improved analysis should eliminate many of the false-positives, and some of the low-risk results. The number of high-risk results should not change. The source code which dealt with modular exponentiation was analyzed in the following configurations:

1. Run Analysis with modulus and power data set tainted/untrusted
2. Sort results into False Positives, Low and High Risk
3. Rerun analysis with white-list to remove some/all of false-positives and low-risk results.
4. Classify new high-risk results into categories based on context.

4.3.2 Classification Stage 1

The results from the experiments are sorted into four categories: false-positives, validation, low-risk and high-risk. The categories are used to build a white-list to further refine the results from the analysis. Results which are classified as false positives are marked due to not being calculated from sensitive data, being marked sensitive as a result of imprecision. This analysis is not flow-sensitive, so results which involve values which are re-computed from sensitive data later in an execution are also considered false-positives. In the baseline analysis, false positives show up due to the fact that a structure or an array is treated as one entity. Fields within that structure or array which have not been computed from sensitive data are reported falsely.

Validation results are branches which lead to exit or error handling code. Validation branches will not execute for valid inputs. For any valid input then, the result of this branch will be the same. The branch may check sensitive data but if the result of the branch is known to an adversary, only trivial information would be gained (i.e. whether or not the data is valid).

Results which are not in the first two categories are then sorted between the low- and high-risk sets of results. The distinction between a low and a high-risk result is on how much information is contained about sensitive data if the outcome of a branch result is known. High-risk means that there is a possibility to leak multiple bits of sensitive data from either a single or repeated execution of the branch. Low-risk results can be identified if the values in the branch condition can be the same across a set of sensitive values. For example, many number libraries have a branch based on the length of a sensitive value, but that value is going to be the same for a range of values. If the number of results is large, it may be helpful to sort the results this way in order to rank results based these categorizations.

If the analysis reports a large number of results, it can be helpful to utilize the classifications done to remove entries from the list. The white-list can be compiled by identifying the variables which propagate and adding them to the white-list. The white-list is often very small yet can eliminate many of the false-positive or low-risk results.

4.3.3 Classification Stage 2

The second stage of classification was done after adding a small set of variables to the white-list. This stage breaks down the high-risk category from the previous stage based on the surrounding context. The first

Library	BP	RP	FP	LR	V	HR
Libgcrypt 1.8.2	64	18	2	10	1	5
mbedTLS 2.9.0	40	25	2	13	3	4
BearSSL 0.5	1	1	0	1	0	0
OpenSSL 1.1.0g						
Reciprocal	32	13	3	2	6	2
Montgomery	34	21	3	5	9	4
Montgomery Const. Time	30	12	0	7	3	2
Montgomery Word	22	14	2	6	5	1

Table 4.2: Result Classifications:BP - Baseline Positives, RP - Reduced Positives(Best Benchmark), FP - False Positive, V - Validation, LR - Low-Risk, V- Validation, HR - High-Risk

criterion is whether or not the branch is within a loop. The second criterion is if the branch is controllable. The high-risk results can then be classified as either, single, repeated, controllable, or loop-controllable. Controllable means that the branch outcome is dependent on input from an untrusted source. In the case of modular exponentiation, the plain-text used in the cryptographic algorithms is untrusted, ignoring the effects of blinding. So a branch would be controllable if there was a comparison between the untrusted data and tainted data such as the power or modulus.

Single branches are branches are not within a loop and are not controllable. Controllable branches are not within a loop. Repeated branches are within a loop and are not controllable. Loop-controllable branches are branches which are within a loop and are controllable.

4.3.4 Experiment Results

In this section results are shown from the library test which had the lowest number of positives while not losing high-risk results. Ideally, the total number of positives decreases and the high-risk results are not sacrificed for that outcome. The looped and controllable results are the ones that are considered high-risk. The summary for all the libraries is given in Table 4.2. After each libraries results are discussed to understand both the false positives and the high-risk results.

All libraries (BearSSL excluded) saw an improvement from adding field-sensitivity, whitelist and/or additional source. The configuration which yielded the best results was including all 3 options. BearSSL did not improve with all 3 configuration options, but all the other libraries saw a reduction in low-risk or false positives. Libgcrypts best configuration did

not include additional source code. Adding extra source code to the tests did not yield the least positives out of all the benchmarks for that library. Libgcrypt had many false-positives removed between the improved and the baseline analysis. The false positives here were due to the multi-precision integer structures having multiple fields containing public data. Many of the branches flagged as vulnerable were falsely reported due to the baseline analysis being not being field-sensitive.

In the mbed TLS library had no changes when the offset was used, so the results here were not due to field-sensitivity issues. BearSSL does not use a structure, but instead just an array to represent multi-precision integers. There were no additional fields so field-sensitivity had no effect on the results for this library either. The false positives however were due to the length being computed from the data within the integers.

OpenSSL showed some reduction in the number of results while maintaining all of the high-risk results. The number of false positives did not change, but the number of low-risk results were removed by adding field-sensitivity. OpenSSL does not modify the multi-precision integers in the files analyzed so many calculations and the flow between them are uncertain.

4.3.5 Libgcrypt High-risk results

Libgcrypt has a multi-precision integer library with a file specifically for modular exponentiation. This library operates on the inputs within this source file, only executing external function calls for other math operations. Unlike mbedTLS and openSSL, the branches are based on the inputs instead of the return value of external functions. This is evident by the fact that adding field-sensitivity alone reduced the number of positives from the baseline. The best result for this library was when white-listing and field-sensitivity were used together.

The first high-risk results comes from the MPN_NORMALIZE call, which is a macro, which scans the exponent pointer and sets the size accordingly. This is done in a branch while directly checking the value of the sensitive data. Once the first non-zero element is found, esize reflects the position of that element. The source is shown in figure 4.3.

This code is listed within the macro `MPN_NORMALIZE` and the input parameters are a pointer d and a size n. The exponent data pointer is passed as d and its corresponding size n. The if branch is then dependent on the sensitive data stored at the pointer. This result is interesting because it is not a constant-time implementation.

```
1  #define MPN_NORMALIZE(d, n)  \
2      do {                     \
3    while( (n) > 0 ) {         \
4        if( (d)[(n)-1] )       \
5      break;                   \
6        (n)--;                 \
7    }                          \
8      } while(0)
```

Figure 4.3: Libgcrypt 1.8.2 - mpi-internal.h lines 113-120

The next four results are within the main processing loop portion of the modular exponentiation code. The main loop of the has a precomputed set of powers and the correct member of that set is selected based on the set bit of the exponent. The first high-risk result from this comparison is shown in line 2 of the abbreviated main loop code shown in Code Fragment 4.4.

The multi-precision integer structure represent large numbers by partitioning each number into multiple *limbs*. Each of these limbs is processed individually in this loop. For each iteration of the loop e is set to the current limb being processed. The *count_leading_zeros* is called to ensure the first bit in the limb is a set bit. Along with the e there is a variable c which tracks how many bits are to be processed in the limb. The value of c changes between each iteration and is dependent on the value of the exponent. Since, e is the data of one limb of the exponent directly, and c is computed directly from the limbs data, these are high-risk variables. Within the main processing loop, the first branch is on e checking if there are no set bits in the limb e (line 3, figure **??**). If this is the case, the process just moves on to the next limb. If there are set bits, in e the rest of the limb is scanned in order to compute the result. If the outcome of this branch is determined, then the value of the entire limb is known and this is a high-risk result.

The operations done within the else are done to select the proper precomputed values based on the value of the limb. The variable c is the position of the first set bit within a limb. The value of c is computed by looking at each bit from a limb in the exponent, making c high-risk. The branch on line 6 (figure 4.5) is in the main processing loop, so it is classified as a looped high-risk result.

At the end of each loop iteration, a non-constant time for-loop runs (Fig 4.6). The for-loop varies with the number of leading zeros for the

```
e = ep[i];
e = (e << c) << 1;
...
for(;;)
  if (e == 0)
    {
      j += c;
      if ( --i < 0 )
        break;

      e = ep[i];
      c = BITS_PER_MPI_LIMB;
    } else ...
```

Figure 4.4: Libgcrypt lines 609-626

```
count_leading_zeros (c0, e);
e = (e << c0);
c -= c0;
j += c0;

e0 = (e >> (BITS_PER_MPI_LIMB - W));
if (c >= W)
  c0 = 0;
else
  {
    if ( --i < 0 ) {
        e0 = (e >> (BITS_PER_MPI_LIMB - c));
        j += c - W;
        goto last_step;
      }
    else
      {
        c0 = c;
        e = ep[i];
        c = BITS_PER_MPI_LIMB;
        e0 |= (e >> (BITS_PER_MPI_LIMB - (W - c0)));
      }
  }
```

Listing 4.1: Libgcrypt lines 635-658

Figure 4.5: Libgcrypt lines 635-658

exponent. The value of j depends on the *c0*, a value derived from the limb of the exponent. This makes j a high-risk variable, and the for loop branch

iterates $j > 0$ meaning that the number of iterations of the loop is not constant time. In this section of the source, there are precautions taken to target some timing channels such as indexing into the precomp array. The conditional set makes it such that for each value of k, the array value is stored. Similarly, to mask the size of the answer stored in *base_u*, the *base_u_size* is computed and set each time, but only the bit mask allows only the true size to be set.

```
1    for (j += W - c0; j >= 0; j--)
2      {
3
4        /*
5          *  base_u <= precomp[e0]
6          *  base_u_size <= precomp_size[e0]
7          */
8        base_u_size = 0;
9        for (k = 0; k < (1<< (W - 1)); k++)
10         {
11           w.alloced = w.nlimbs = precomp_size[k];
12           u.alloced = u.nlimbs = precomp_size[k];
13           u.d = precomp[k];
14
15           mpi_set_cond (&w, &u, k == e0);
16           base_u_size |= ( precomp_size[k] & (0UL - (k == e0
                 )) );
17         }
18
19       w.alloced = w.nlimbs = rsize;
20       u.alloced = u.nlimbs = rsize;
21       u.d = rp;
22       mpi_set_cond (&w, &u, j != 0);
23       base_u_size ^= ((base_u_size ^ rsize)  & (0UL - (j !=
             0)));
24
25       mul_mod (xp, &xsize, rp, rsize, base_u, base_u_size,
26                 mp, msize, &karactx);
27       tp = rp; rp = xp; xp = tp;
28       rsize = xsize;
29     }
```

Figure 4.6: Libgcrypt mpi-pow.c lines 667-695

The while loop shown in figure 4.7 is done after the main processing loop. The value of j is related to the number of non-set bits in the last processed limb making it a high-risk variable. The branch is the condition for a loop,

so this is a looped high-risk result.

```
1  while (j--)
2    {
3      mul_mod (xp, &xsize, rp, rsize, rp, rsize,
4                               mp, msize, &karactx);
5      tp = rp; rp = xp; xp = tp;
6      rsize = xsize;
7    }
```

Figure 4.7: Libgcrypt mpi-pow.c lines 702-707

4.3.6 mbedTLS 2.9.0

The big number library for mbedTLS is a single source file. For the baseline results, the file was modified to only include the source required to compile and analyze the modular exponentiation code. The least amount of positives was found when combining, field-sensitivity, white listing and including the entire big number source. The high risk results in this library are branches which look at a single bit in the exponent.
The mbedTLS library had no change in the number of positives with the field-sensitivity alone. However combining field-sensitivity with the full library source code improved results.
In the modular exponentiation code, there is a reduction of the base a if it is greater than the modulus n. The modulus n is marked to be tainted and a is the multi-precision integer that represents the base. The base is the user input, and is untrusted. This branch on line 1673 of the source file (*bignum.c*) directly compares the value of a to the value of n.

```
1  if( mbedtls_mpi_cmp_mpi( A, N ) >= 0 )
2      MBEDTLS_MPI_CHK( mbedtls_mpi_mod_mpi( &W[1], A, N ) );
3  else
4      MBEDTLS_MPI_CHK( mbedtls_mpi_copy( &W[1], A ) );
```

Figure 4.8: mbedTLS 2.9.0 - bignum.c lines 1672-1676

The 3 looped results reside in the loop which inspects the bits of the exponent. The data for the limbs containing the exponent is located at *E-¿p*. Each iteration through the loop looks at one bit of the exponent directly, making *ei* a high-risk variable in this loop. The first branch on *ei* on line 1736 is a high-risk looped branch to skip leading zeros, so that each

window starts on a set bit. The second branch on line 1739 runs after a full window has been processed and the subsequent bits are zero.

4.3.7 BearSSL 0.5

BearSSL had no positives when setting the exponent and modulus within the modular exponentiation code to be the tainted values. The library claims to have constant-time implementations. The mod_pow function itself had 0 positives, since there were no secret dependent branches. The library contains very small files so additional files were linked to get source which closer resembled the source analyzed in the other libraries. The added source files included the definitions of the functions called in the mod_pow function.

A total of 1 positive was found with the additional source code. The branch is found in the conversion into Montgomery form. The positive is shown on line 6 of figure 4.10.

In this library, no structure represents the big integers, instead an array serves the purpose. The first element in the array is used to compute the size and all subsequent elements in the array are the data. The low number of results here is not due to the field sensitivity, but the ability to set the tainted variables by function context. The only variables set to be tainted in the analysis are e and m within the *br_i32_modpow* function.

4.3.8 OpenSSL1.1.0g

The number of positives was least when utilizing the field-sensitive analysis with a whitelist and the full big number library source. More than half the original number of positives were eliminated when using the best configuration. There were still remaining false positives, which are a result of pointers to different data-types pointing to the same node. In this library, the exponent, the plain-text and the modulus and result pointers all were represented by the same node. This means that even though in practice those pointers point to different memory locations, all were treated as the same memory location. The nodes were separate at the beginning of the analysis, but the points-to analysis deemed these nodes as equivalent, and merged them together in the memory graph.

There are 5 methods for carrying out the modular exponentiation defined in the OpenSSL. There is the reciprocal method, and 3 variations of the Montgomery method, and then a simple sliding-window method. The last method is unreachable from the *BN_mod_exp* function, so no results are

included for that method.

Reciprocal Method

The reciprocal method has two looped high-risk positives both leaking data from the exponent used in the modular exponentiation. The variables are the exponent p and the modulus m. Both of these structures have the data elements are located as the first element in the structure. The tainted variables for this function are p 0 and m 0. The main processing loop, shown in figure 4.11, traverses the bits of p and looks for the first set bit, resulting in the first looped positive. The function *BN_is_bit_set* is from *bn_lib.c* and is linked with *bn_exp.c* for the source definition. This means that since p 0 is tainted, the *BN_is_bit_set* function result is derived from p 0. From the function definition in fig. 4.12, the data array from the bignum structure is indexed using some transformation of the input parameter n and then masks all but one bit. This function leaks 1 bit of p at a time, but is used within a loop, where the input parameter is *wstart* and changes with the loop iteration causing more bits of p to be leaked as the loop executes.

This result is still reported if the *bn_lib* source file is not linked with the *bn_exp.c* file. When the function definition is unreachable, all reachable sources from the arguments flow to the return values and other mutable values. In this case this means that since p 0 is tainted, the return value from the *BN_is_bit_set* function is also tainted.

Within the same processing loop, the second high-risk looped result can be found, line 6 in fig. 4.13. The analysis reports this result for the same reason as the previous result. The analysis results are only tracking explicit flow, but an important implicit flow result can be followed from this positive. Since the loop variable i flows to the window end variable *wend* when this branch condition is true, there is implicit flow between the data in p and the *wend* variable. The subsequent for loop then ranges from 0 to j a value computed simply from *wend*. The number of iterations for the loop from 0 to j is then a function of the value of p.

Montgomery Method

In the high-risk classifications, there is the category of controllable branches. These are when there is a branch which the outcome is dependent both on user-input as well as secret data. As with the reciprocal method, the exponent and modulus, p 0 and m 0 respectively, are tainted. The untrusted user specified data is a 0 which is another bignum structure. Line 1 in fig. 4.14 is controllable, because there is a comparison operation done between the plain-text and modulus. Assuming a 0 is controllable, then it could be possible to change a 0 enough to discover m 0. This

branch is done to satisfy the assumption that $aa < m$ for the rest of the processing.

Line 2 of fig. 4.14 calls a function that is not linked with the analyzed source. One of the results reported from the analysis is line 374 of *bn_exp.c*. This line is reported due to flow from *m* to *val[0]* from the undefined function *BN_nnmod*. Depending on the branch taken the value of *aa* is equivalent to *val[0]* and thus the branch on the *BN_to_montgomery* call is tainted due to the result of that branch being dependent on tainted data.

The two high-risk looped results are from code identical to the reciprocal method branching on set bits in p within the main processing loop. These lines for the Montgomery method are located at 416 and 437 respectively in *bn_exp.c*.

Constant-Time Montgomery Method

There is one controllable positive for the constant-time Montgomery method. This branch is similar to the one found in the Montgomery method, but instead of calling *bn_nnmod* the normal *bn_mod* is called. This result is seen in fig. 4.15.

There are still function calls to *BN_is_bit_set*, but not as branch conditions. Instead the set bits of the window are extracted, and passed as a parameter to select the correct value from the pre-computed table of powers.

It is possible that there are true positives within the source code given in fig. 4.16. This code defines the *bn_gather5_t4* function, but that source was not analyzed. The only analyzed source files were *bn_exp.c* and *bn_lib.c*.

Montgomery Word Method

Within the Montgomery word method, 1 high-risk looped branch is reported. The high-risk looped branch is the same branch from the other methods involving the BN_is_bit_set function on the exponent *p*. This positive is found on line 1 of figure 4.17.

```
while( 1 )
{
    if( bufsize == 0 )
    {
        if( nblimbs == 0 )
            break;

        nblimbs--;

        bufsize = sizeof( mbedtls_mpi_uint ) << 3;
    }

    bufsize--;

    ei = (E->p[nblimbs] >> bufsize) & 1;

    /*
      * skip leading 0s
      */
    if( ei == 0 && state == 0 )
        continue;

    if( ei == 0 && state == 1 )
    {
        /*
          * out of window, square X
          */
        MBEDTLS_MPI_CHK( mpi_montmul( X, X, N, mm, &T ) );
        continue;
    }

    /*
      * add ei to current window
      */
    state = 2;

    nbits++;
    wbits |= ( ei << ( wsize - nbits ) );

    if( nbits == wsize )
    {
        /*
          * X = X^wsize R^-1 mod N
          */
        for( i = 0; i < wsize; i++ )
            MBEDTLS_MPI_CHK( mpi_montmul( X, X, N, mm, &T )
                );

        /*
          * X = X * W[wbits] R^-1 mod N
          */
        MBEDTLS_MPI_CHK( mpi_montmul( X, &W[wbits], N, mm, &
            T ) );

        state--;
        nbits = 0;
        wbits = 0;
    }
}
```

```
1  void
2  br_i32_to_monty(uint32_t *x, const uint32_t *m)
3  {
4    uint32_t k;
5
6    for (k = (m[0] + 31) >> 5; k > 0; k --) {
7      br_i32_muladd_small(x, 0, m);
8    }
9  }
```

Figure 4.10: BearSSL 0.5 - br_i32_tmont.c

```
1  for (;;) {
2      if (BN_is_bit_set(p, wstart) == 0) {
3          if (!start)
4              if (!BN_mod_mul_reciprocal(r, r, r, &recp, ctx))
5                  goto err;
6          if (wstart == 0)
7              break;
8          wstart--;
9          continue;
10     }
11     ...
12 }
```

Figure 4.11: OpenSSL 1.1.0g - bn_exp.c lines 250 - 259

```
1  int BN_is_bit_set(const BIGNUM *a, int n)
2  {
3      int i, j;
4
5      bn_check_top(a);
6      if (n < 0)
7          return 0;
8      i = n / BN_BITS2;
9      j = n % BN_BITS2;
10     if (a->top <= i)
11         return 0;
12     return (int)(((a->d[i]) >> j) & ((BN_ULONG)1));
13 }
```

Figure 4.12: OpenSSL 1.1.0g - bn_lib.c lines 741 - 753

```
for(;;){
  ...
        for (i = 1; i < window; i++) {
            if (wstart - i < 0)
                break;
            if (BN_is_bit_set(p, wstart - i)) {
                wvalue <<= (i - wend);
                wvalue |= 1;
                wend = i;
            }
        }

        /* wend is the size of the current window */
        j = wend + 1;
        /* add the 'bytes above' */
        if (!start)
            for (i = 0; i < j; i++) {
                if (!BN_mod_mul_reciprocal(r, r, r, &recp,
                    ctx))
                    goto err;
            }
   ...
}
```

Figure 4.13: OpenSSL 1.1.0g - bn_exp.c lines 250-297

```
if (a->neg || BN_ucmp(a, m) >= 0) {
    if (!BN_nnmod(val[0], a, m, ctx))
        goto err;
    aa = val[0];
} else
    aa = a;

...
if (!BN_to_montgomery(val[0], aa, mont, ctx))
    goto err;               /* 1 */
```

Figure 4.14: OpenSSL 1.1.0g - bn_exp.c lines 363-368

```
if (a->neg || BN_ucmp(a, m) >= 0) {
    if (!BN_mod(&am, a, m, ctx))
        goto err;
    if (!BN_to_montgomery(&am, &am, mont, ctx))
        goto err;
} else if (!BN_to_montgomery(&am, a, mont, ctx))
    goto err;
```

Figure 4.15: OpenSSL 1.1.0g - bn_exp.c lines 752-758

```
for (wvalue = 0, i = bits % 5; i >= 0; i--, bits--)
    wvalue = (wvalue << 1) + BN_is_bit_set(p, bits);
bn_gather5_t4(tmp.d, top, powerbuf, wvalue);
```

Figure 4.16: OpenSSL 1.1.0g - bn_exp.c lines 852-854

```
if (BN_is_bit_set(p, b)) {
    next_w = w * a;
    if ((next_w / a) != w) { /* overflow */
        if (r_is_one) {
            if (!BN_TO_MONTGOMERY_WORD(r, w, mont))
                goto err;
            r_is_one = 0;
        } else {
            if (!BN_MOD_MUL_WORD(r, w, m))
                goto err;
        }
        next_w = a;
    }
    w = next_w;
}
```

Figure 4.17: OpenSSL 1.1.0g - bn_exp.c lines 1200-1214