1 Introduction

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 1 to illustrate this concern.

Algorithm 1 Simple Information Flow

k = sensitive information
$d.x = k + 1$
$d.y = 0$
$a = -1$
if $d.y == k/2$ **then**
    $a = 4$
end if

In this simple example k is confidential data, through information flow we 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.

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 1, it must be possible to identify from the source the location of d, and subsequently the location 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.

Algorithm 2 Square and Multiply Timing Channel

```
for (i = 1; i < bits; i++) {
    if (!BN_sqr(v, v, ctx))
        goto err;
    if (BN_is_bit_set(p, i)) {
        if (!BN_mul(rr, rr, v, ctx))
            goto err;
    }
}
```

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 [1]. 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.

3 Improving Field Sensitivity in Taint Analysis

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

The first component of the analysis which identifies the information flows of the program, works on the instruction level. For each instruction a set of flows will be identified, called a Flow Record. Each instruction will generate an explicit and implicit flow record, which yield different results depending

on the type of analysis run. The analysis attempts to start from the main function, but if it does not exist all functions will be analyzed as the start point. Any function which calls other functions within its definition, is analyzed in its own context, so information can flow through the arguments between two functions. For each function context, all instructions are parsed to identify operands and operation. The operands are divided into source and sink operands and based on the operation. The flow is added to the corresponding set based on whether or not flow is explicit or implicit. This section has not been modified in the improved version of the analysis.

3.2 Constraint Generation

After the information flow capture component is finished, there are a set of flow records for each function context and this stage will generate the corresponding constraints for each flow record. The changes to improve the baseline analysis are mostly done in this stage. Once the flows through the program are identified, each constraint which makes up the set, is generated by iterating through the flows. A constraint is added to the set constraining the sink element in relation to the source element.

The first time an instruction is encountered as either a source or a sink, a constraint element would be generated. For most instructions, just one constraint element should be generated as the instruction serves as a source or a sink to only one value in the program. For operations which load from compound data type, such as a class, struct, or array, the baseline generated one constraint element per data type instead of one for each field within the type. This lead to imprecision particularly when sensitive data flowed to a part of the compound type, the field it flowed to would be tainted as well as all other field which shared the same memory location. The GetElementPtr (GEP) instructions in LLVM is one instruction in particular that is used when getting a field at some offset from a base address. The improved analysis treats this instruction differently than others to achieve a more precise result.

GEP instructions are the intermediate level instruction used when addressing a field within a compound type. These types are treated as pointer types and all fields are offsets from them, so the points-to analysis is used to create elements unique to each instance of a compound type. Each unique compound type is captured by recognizing the GEP instruction source or sink, and finding the corresponding memory location it addresses with the help of the points-to analysis. Once captured, if the data layout can be determined for the type, a constraint element is generated for each field in the type at the offset assigned by the data layout information stored in the points-to analysis memory node. After this set of elements is created, the field offset is an operand within the instruction is used to pick the corresponding element to be constrained. The rest of the instructions are handled

the same as the baseline. After iterating through the flow record, with this change, the set of constraints generated has been changed to be specific enough to have flow between elements within a compound type without tainting any accompanying fields.

3.3 Solving Set of Constraints

Given the set of constraints for the program being analyzed, the smallest solution where all the sinks are at least as large as their sources is found, otherwise known as a least solution. From the solution each sink operand 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, and an attacker may be able to exploit the computed confidential data to learn information or reconstruct the original confidential data.

Consider Algorithm 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 Algorithm 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. Algorithm 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.

Algorithm 3 Public and private data in structure

```
struct AES_KEY_t {
  unsigned * rd_key;
  int rounds;
};

typedef AES_KEY_t AES_KEY;

void AES_enc(char * in, char * out, AES_KEY) {
  unsigned * rk = key->rd_key;
  if (key->rounds > 10) {
    ...;
  }
}
```

3.4 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 Algorithm 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.

Algorithm 4 Validation Source Code

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

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.

Algorithm 5 High-Risk and Low-Risk Branches

```
#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;
```

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

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.

Algorithm 6 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.

Algorithm 6 Creating constraint elements for each field in a type

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

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 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 Algorithm 7. Given the node, the data layout and associated structure layout can be found and then the index can be converted to a byte offset.

Algorithm 7 Calculating byte offset from index

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

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 arugments or the return value. When the field-sensitivity was implemeted, 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 senitive data to all sinks from that function. After the field sensitivity change, the pointer and its data are constrained seperately, 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

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 whitelist 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 whitelist 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 whitelist can be compiled by identifying the variables which propogate and adding them to the whitelist. The whitelist 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 classsifcation was done after adding a small set of variables to the whitelist. This stage breaks down the high-risk category from the previous stage based on the surrounding context. The first 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 plaintext 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 controllabe. 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 Experimental Results

In this section results from the baseline and the improved analysis are compared. The source which contained the the modular exponentiation functions were compiled with only the necessary header files. This meant that some calls were only to functions which were declared but not yet defined.

In order for the improved analysis to in fact be an improvement over the baseline system, the analysis will have to reduce the number of results. Reducing the results should not be at the cost of high-risk results either. After the first stage the classification results without the improvement are shown in Table 1 and with the improvment the results are shown in Table 2. For some libraries, the number of false positives is high, such as in Libgcrypt. Many of these false positives are due to lack of field sensitivity. OpenSSL has multiple implementations of the modular exponentiation. The results were separated to show the results that corresponded to that implementation.

Library	TP	FP	V	LR	HR
Libgcrypt 1.8.2	68	36	1	17	14
mbedTLS 2.9.0	25	8	0	13	4
BearSSL 0.5	10	9	1	0	0
OpenSSL 1.1.0g					
Reciprocal	24	4	6	12	2
Montgomery	34	4	11	15	4
Montgomery Const. Time	30	0	12	16	2
Montgomery Word	22	3	11	3	4
Simple	22	2	6	12	2

Table 1: Stage 1 Classification - Field Insensitive

With the field-sensitive analysis, the constraints generated were now able to include offsets and the total number of results were fewer. The analysis changes had a significant effect in reducing the number of false positive results for Libgcrypt, but only small effects in the other libraries. The most important part is that the high-risk category had the same number of results even with the reduction in total number of results. The number of high-risk results staying the same as a result of the field sensitivity change, that alone actually loses some of the high-risk results. OpenSSL does much of its work in functions external to the modular exponentiation code, meaning that there is little to no manipulation of the data directly in the source analyzed.

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 mutate the multi-precision integers in the files analyzed so many calculations and the flow between them are uncertain.

Library	TP	FP	V	LR	HR
Libgcrypt 1.8.2	34	2	1	17	14
mbedTLS 2.9.0	25	8	0	13	4
BearSSL 0.5	10	9	1	0	0
OpenSSL 1.1.0g					
Reciprocal	13	4	6	1	2
Montgomery	25	4	11	6	4
Montgomery Const. Time	21	0	12	7	2
Montgomery Word	20	3	11	2	4
Simple	11	2	6	1	2

Table 2: Stage 1 Classification - Field Sensitive

Single	4
Looped	10
Controllable	0
Both	0

Table 3: High-Risk Result Classification

4.3.5 Libgcrypt High-risk results

Libgcrypt 14 high-risk results are broken down further as given in table

There were no controllable branches which are the most severe, allowing for the users input text to be directly compared to the sensitive data. At the start of the modular exponentiation, the first looped positive is found. The exponent is given in the multi-precision structure, and one of the first operations done is to find the size through the normalization code given in the code fragment 1.

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

Code Fragment 1: Libgcrypt 1.8.2 - mpi-internal.h lines 113-120

This code is listed within the macro `MPN_NORMALZE` 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.

The modulus is adjusted based on its value as well through the code shown in Algorithm 2. In this snippet, the number of leading zeros is stored, and leaks information about the number of zero bits at the start of the modulus. This *mod_shift_cnt* is used later in lines 715, 738, and 751. All cases of these branches are done outside of loops, so the classification for this result is high-risk single.

```
# define count_leading_zeros(count, x) \
    __asm__ ("clz␣%0,%1"                \
       : "=r" ((USItype)(count)) \
       : "r" ((USItype)(x)))

count_leading_zeros (mod_shift_cnt, mod->d[msize-1]);
if (mod_shift_cnt)
  _gcry_mpih_lshift (mp, mod->d, msize, mod_shift_cnt);
else
  MPN_COPY( mp, mod->d, msize );
```

Code Fragment 2: Libgcrypt 1.8.2 - mpi-pow.c lines 483-487

The next pre-processing step compares the base and the modulus sizes. The sizes themselves are not tainted. The tainted variabl here is mp, a pointer to the modulus data. In the analysis, the *divrem* function is not included source code, so the mutable parameters, in this case *bp* is tainted as a result. The normalize code is the same seen earlier, with a while loop which branches on the size of the data, and a if which branches based on the data located at the pointer. In this case the pointer is *bp* considered a high-risk variable due to having flow from the modulus data. The *MPN_NORMALIZE* line is classified as high-risk.

```
  if (bsize > msize)
    {
      ...
      MPN_COPY ( bp, base->d, bsize );
      _gcry_mpih_divrem( bp + msize, 0, bp, bsize, mp, msize );
      bsize = msize;
      ...
      MPN_NORMALIZE( bp, bsize );
    }
```

Code Fragment 3: Libgcrypt 1.8.2 - mpi-pow.c lines 491-507

The next two 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.

The multi-precision integer structure represent large numbers by partioning 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 porcessing loop, the first branch is on e checking if there are no set bits in the limb e. 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.

```
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 ...
```

Code Fragment 4: Libgcrypt lines 609-626

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. This branch is in the main processing loop, so it is classified as a looped high-risk result.

```
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)));
    }
}
```

Code Fragment 5: Libgcrypt lines 635-658

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

```
for (j += W - c0; j >= 0; j--)
  {

    /*
      *  base_u <= precomp[e0]
      *  base_u_size <= precomp_size[e0]
      */
    base_u_size = 0;
    for (k = 0; k < (1<< (W - 1)); k++)
      {
        w.alloced = w.nlimbs = precomp_size[k];
        u.alloced = u.nlimbs = precomp_size[k];
        u.d = precomp[k];

        mpi_set_cond (&w, &u, k == e0);
        base_u_size |= ( precomp_size[k] & (0UL - (k == e0)) );
      }

    w.alloced = w.nlimbs = rsize;
    u.alloced = u.nlimbs = rsize;
    u.d = rp;
    mpi_set_cond (&w, &u, j != 0);
    base_u_size ^= ((base_u_size ^ rsize)  & (0UL - (j != 0)));

    mul_mod (xp, &xsize, rp, rsize, base_u, base_u_size,
             mp, msize, &karactx);
    tp = rp; rp = xp; xp = tp;
    rsize = xsize;
  }
```

The while loop shown in fragment 6 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.

```
while (j--)
  {
    mul_mod (xp, &xsize, rp, rsize, rp, rsize,
                          mp, msize, &karactx);
    tp = rp; rp = xp; xp = tp;
    rsize = xsize;
  }
```

Code Fragment 6: Libgcrypt lines 702-707

There are two more high-risk results after the main processing is over, involving the result data on lines 740 and 756. This data has flow from the modulus and exponent and so it is considered sensitive. The *rsize* values are also considered sensitive due to the flow between parameters in the *mul_mod* calls. Both of the branches from the *MPN_NORMALIZE* macro are reported and are considered both high-risk looped results.

4.3.6 mbedTLS 2.9.0

The mbedtls library had no change in the number of positives when including the offset. 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.

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

Code Fragment 7: mbed TLS 2.9.0 - bignum.c lines 1673-1676

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.

```
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;
}
```

```
}
```

Code Fragment 8: mbed TLS 2.9.0 - bignum.c lines 1717-1773

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 be constant-time implementations. The library is small and modular, so the 32 bit big integer source codes were all linked. There were 10 positives from the analysis, all of which are elimnated by the whitelist.

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 context. The only variables set to be tainted in the analysis are e and m within the *br_i32_modpow* function.

From the modular exponentiation source alone, there are no positives. All of the positives which were eliminated using the whitelist were based on the length from the first element in the arrays.

4.3.8 OpenSSL1.1.0g

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 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, 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.

```
for (;;) {
    if (BN_is_bit_set(p, wstart) == 0) {
        if (!start)
            if (!BN_mod_mul_reciprocal(r, r, r, &recp, ctx))
                goto err;
        if (wstart == 0)
            break;
        wstart--;
        continue;
    }
   ...
}
```

Code Fragment 9: OpenSSL 1.1.0g - bn_exp.c lines 250 - 259

```
int BN_is_bit_set(const BIGNUM *a, int n)
{
    int i, j;

    bn_check_top(a);
    if (n < 0)
        return 0;
    i = n / BN_BITS2;
    j = n % BN_BITS2;
    if (a->top <= i)
        return 0;
    return (int)(((a->d[i]) >> j) & ((BN_ULONG)1));
}
```

Code Fragment 10: OpenSSL 1.1.0g - bn_lib.c lines 741 - 753

Within the same processing loop, the second high-risk looped result can be found. 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.

```
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;
            }
    ...
}
```

Code Fragment 11: OpenSSL 1.1.0g - bn_exp.c lines 250-297

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.

This positive is controllable, because there is a comparison operation done between the plaintext 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.

```
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 */
```

Code Fragment 12: OpenSSL 1.1.0g - bn_exp.c lines 363-368

Line 2 of this code fragment 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 porcessing 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.

```
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;
```

Code Fragment 13: OpenSSL 1.1.0g - bn_exp.c lines 752-758

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.

```
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);
```

Code Fragment 14: OpenSSL 1.1.0g - bn_exp.c lines 852-854

It is possible that there are true positives within the source code which 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*.

Mongomery Word Method

Within the montgomery word method, there are 3 controllable branches (2 of which are looped), and 1 high-risk looped branch. The high-risk looped branch is the same branch from the other methods involving the BN_is_bit_set function on the exponent p.

The main procesing loop is shown, a is the untrusted plain text, and p 0 and m 0 are the tainted values. Within this loop, there are function calls to *BN_TO_MONTGOMERY_WORD* and *BN_MOD_MUL_WORD* which are macros that call functions that are not linked with the analyzed source. Again, the analysis assumes flow between mutable parameters if the function defintion is not provided. This results in there being an assumed flow between w and a and m. Branches on lines 1182 and 1202 are controllable since a is an untrusted input and m is tainted data.

The final controllable but not looped result does one final check on =w=. This result is high-risk due to the flow from =m= and =a= to =w=.

```
  w = a;                        /* bit 'bits-1' of 'p' is always set */
  for (b = bits - 2; b >= 0; b--) {
      /* First, square r*w. */
      next_w = w * w;
      if ((next_w / w) != 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 = 1;
    }
    w = next_w;
    if (!r_is_one) {
        if (!BN_mod_mul_montgomery(r, r, r, mont, ctx))
            goto err;
    }

    /* Second, multiply r*w by 'a' if exponent bit is set. */
    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;
    }
}

  /* Finally, set r:=r*w. */
  if (w != 1) {
      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;
      }
  }
```

Code Fragment 15: OpenSSL 1.1.0g - bn_exp.c lines 1178-1227

Simple Method

The positives for the *BN_mod_exp_simple* algorithm are the same as the ones reported in the reciprocal and montgomery methods. In the main procesing loop there are two branches that depend on the function *BN_is_bit_set*. The input to the function is the exponent p which is a tainted value. These positives can be found on lines 1312 and 1332 of *bn_exp.c*.

5 Related Works

6 Conclusion

References

[1] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 555–576. Springer, 2017.