CtChecker: a Highly Precise and Efficient Static Information Analysis for Detecting Non-Constant Time Code

Adam Mohammed
The Pennsylvania State University
aqm5498@psu.edu

Ernest DeFoy III
The Pennsylvania State University
edd5103@psu.edu

Danfeng Zhang
The Pennsylvania State University
zhang@psu.edu

ABSTRACT

Timing channel attacks are emerging as real-world threats to computer security. While cryptosystems have been the major targets of timing attacks, recent attacks such as Spectre and Meltdown show that non-constant time code, when combined with other attack vectors such as speculative execution, can reveal arbitrary memory from the victim's process. In cryptosystems, an effective countermeasure against timing attacks is the constant-time programming discipline. However, strictly enforcing the discipline manually is both time consuming and error prone. Recent efforts on identifying non-constant code fragments verify a 2-property, meaning that with identical public inputs, two runs of a program will produce identical control flows, memory accesses, etc. However, at least implicitly, such tools consider 2 runs of the program, resulting in either poor performance or key information (e.g., loop invariants) across 2 runs being lost.

In this paper, we build CtChecker, a static sound analysis for identifying non-constant code. Under the hood, CtChecker verifies a 1-property, namely program dependence, making it more efficient than existing constant-time verifiers. For precision, CtChecker is both field-sensitive and context-sensitive, and it supports declassification. Somewhat surprisingly, CtChecker reports fewer false positives in cryptosystems compared with existing tools, since the latter suffers from lost information such as loop invariants. Evaluation on multiple real-world cryptosystems shows that CtChecker analyzes over 40K lines of source code in half an hour, and its false-positive rate is low.

CCS CONCEPTS

• **Security and privacy → Information flow control**;

ACM Reference Format:
Adam Mohammed, Ernest DeFoy III, and Danfeng Zhang. 2019. CtChecker: a Highly Precise and Efficient Static Information Analysis for Detecting Non-Constant Time Code. In *Proceedings of Computer and Communications Security (CCS'19).* ACM, New York, NY, USA, **??** pages. https://doi.org/0000001.0000001_2

1 INTRODUCTION

Timing channels 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 timing channels to compromise the assumptions of these safe communication methods. Timing channels exist in many forms and are often the result of implementation details which are not available in theoretical proposals of a protocol.

In cryptosystems, an effective countermeasure against timing attacks is the constant-time programming discipline. While a strict enforcement of this discipline will likely rule out timing attacks, doing so is also infeasible for a couple of reasons: (1) identifying non-constant time code is often an error-prone and manual process, (2) real-world code typically contains intentional non-constant fragments that does not leak crucial information.

Many methods have emerged as a response to attempt to automate the process of identifying timing channels. 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 timing channels, and mitigate them as necessary. The problem becomes identifying the vulnerabilities. Static analyses exist to be able to identify these vulnerabilities.

Static label-based information Flow Control (IFC) provides strong security guarantees for untrusted code, but are often believed to provide a non-negligible rate of false alarms for real-world software. 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 timing 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 *DSA*, a points-to analysis to handle dynamically allocated memory [**?**]. The information-flow analysis is heavily reliant on the points-to analysis to track the information flow of practical programs. The precision of the information-flow analysis is influenced by how it integrates with the points-to analysis.

The static information flow analysis 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 cryptosystems, 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 points-to analysis. 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. For example, in Libgcrypt the number of results was reduced from 64 to 24 total results. Of the 64 baseline results there were 5 classified as high-risk and in the improved analysis the high-risk results were still present. The reduced number of results came from eliminated false positives.

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 figure **??** to illustrate this concern.

In this simple example, k is confidential data, and 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 since its value does not rely on the confidential data. The flow between k and $d.x$ is explicit flow since the value of $d.x$ is computed from the

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

Figure 1: Example of Explicit and Implicit Flow

confidential data. As the analysis continues, any value computed from $d.x$ will also be treated as confidential data. There is also implicit flow from k to a, 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. 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.

Additionally, information flow analyses are capable of finding hardware based vulnerabilities by analyzing the source. Information flow has been used to detect leakage of confidential data through side channels both in the source and the hardware (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 sufficient once pointers are involved. For example, in Algorithm **??**, 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 remain known, even if the value of d changes (it points to another location in memory). In addition to providing spatial information, the points-to analysis used in this research, DSA, 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 (units of memory) that are able to be the target of a pointer. Data structures identified in program source have one node for each 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. However, this leads to an over-approximation of the resulting tainted values, which will

```
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: Square and Multiply Timing Channel

be discussed later in this paper. The points-to analysis used also has a method of determining the validity of the type information.

2.3 Timing Channels in Cryptosystems

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 can be measured due to differences in each methods execution time. Though timing based, this attack can be observed in the source code of OpenSSL's previous implementation of sliding window exponentiation, shown in Algorithm **??**.

The for-loop here iterates through the number of bits in the confidential power p, each time by 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 since a multiplication is 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. If one can determine the locations of the multiplies and the length of time expected between each multiplication, the key can be rebuilt, which 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.

3 CTCHECKER DESIGN

In this section, we describe the design details of CtChecker.

In this work, adjustments were made to improve the precision of a baseline taint analysis. The baseline analysis was proposed by Moore et al [**?**]. Adding field-sensitivity and adding a whitelist both improved precision. Adding field-sensitivity aided in more accurately representing flow within structures and arrays. The baseline could not track flow to members of a structure, only flow to the structure itself. The field-sensitive analysis is the major change to the baseline system. This section will outline the three sections which make up the analysis. The information flow capture parses the IR code to identify the values from which information flows. The constraint generation is done by interpreting the captured flows and creating constraints. Lastly, to identify secret-dependent branches, the least solution is found and branch conditions are checked to see if sensitive information was used to compute the condition.

3.1 Generating Information Flows from IR Code

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 change, 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 or sinks.

For each LLVM value, there are 3 elements which can be constrained. There are 3 transformation functions which return the element to be constrained for that value. For any operation, zero or more of the three functions (V, D, R) may be used in generating a constraint for the instruction. Forming a constraint requires a source constraint element and a sink constraint element. The flows identified from the instructions create a constraint by adding a rule, which states that the source constraint element flows to the sink constraint element.

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

The baseline analysis used these same transformations, but the addition of field sensitivity changed the behavior of the functions D and R, and the memory nodes 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 *DSA* 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.

The R transformation is similar to the D transformation in that it returns the constraint element relevant to the memory addressed, but the R transformation is used in the case of external functions or source. The R transformation is the more conservative transformation, returning all possible constraint elements tied to a pointer, so the offset and size are not required like they are for the D transformation. In the baseline, the D transformation did not exist, thus this transformation was added to achieve field-sensitivity.

For most IR instructions, 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 *DSA*. The store, load and GetElementPtr (GEP) instructions are the most important for addressing the issue of field sensitivity. The rules for how these are constrained are given in figure **??**.

The load and the store instructions are the instructions which changed the most from the baseline. In the baseline, these rules were the same, but the offset was not considered in the D transformation. The store and load instructions do not have an offset as a operand in their instructions. The offset is found by examining the IR instruction which yielded the operand. If the operand is a pointer computed from a structure or an array, then the pointer

```
<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: $V(\text{value}) \rightarrow D(\text{pointer, offset})$
Implicit: $V(PC) \cup V(\text{pointer}) \rightarrow D(\text{pointer, offset})$

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

Figure 3: LLVM Memory IR Flow Rules

will have been calculated using the GEP instruction. By including the GEP instruction in the sources for the load (and the sink for the store), the instruction can be analyzed and the index can be accessed.

3.1.1 Constraint Generation. For operations which do not modify memory, the constraints are generated such that $V(inputs) \rightarrow V(outputs)$. 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 **??** 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 4: Store Constraint Example

The D transformation utilizes the points-to analysis to retrieve the correct memory node. The memory node stores type information for any types accessed from that node in the IR code. For stack arrays and structures, the type information from the points-to analysis is used to create individual elements for each of the members accessible from that structure/array. Thus the offset must be passed to D along with the pointer to select the correct element. For structures, the instructions only give an index, so the byte offset is computed using the data layout information that the points-to analysis uses to create its type information.

In figure **??** 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.

3.1.2 Factors Affecting Field-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 Information Flow 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. Given a set of initial sensitive values, any sink which has a value greater than or

equal to that value, is considered sensitive. In the case of cryptosystems, 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 **??** 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. In figure **??**, 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 **??** 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 5: 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-risk or low-leakage sets, 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 **??**, 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. To conclude stage 1, if there are results which are reported that leak non-sensitive data, they may be whitelisted. The whitelist allows results to be ignored, and not propagate taint as a result in the later stages.

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.

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

Figure 6: Error-handling Source Code

```
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)
9
10 MPN_NORMALIZE(ep, esize);
11 if (esize * BITS_PER_MPI_LIMB > 512)
12   W = 5;
```

Figure 7: High-Risk and Low-Risk Branches

Figure **??** 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 by handling LLVM's pointer instruction differently than other instructions. The pointer instruction in the IR is the GEP 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 **??** 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.

```
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 8: Creating constraint elements for each field in a type

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 figure **??**. 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 9: 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 **??** show the effects of different features on the number of reported secret-dependent branch warnings. 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 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 implemented modular exponentiation functions were analyzed in the following configurations:

(1) Baseline Analysis
(2) Baseline with field-sensitivity changes only
(3) Baseline with white-list only
(4) Baseline with full library source code
(5) Baseline with whitelist and field-sensitivity
(6) Baseline with field-sensitivity, whitelist and full library source code low-risk results.

4.3.2 Potential Sources of Unsoundness. For these experiments, the results given may not be all the results possible from the source analyzed due to underlying assumptions. Since the full source code was not analyzed, there are function calls which are defined in files which are not compiled and linked. One assumption made for unknown functions is that the flow only exists between the parameters. There is a possibility for this assumption to break down if, for example, pointers are modified to point to something which is not a value provided in the parameter. Another potential cause of missing results is due to only considering explicit flows. The analysis is able to handle both explicit and implicit flow, but currently they are only able to be run independently. The experimental results shown only consider explicit flows.

4.3.3 Performance. The performance running times of the baseline is compared with the best performing improved analysis in Table **??**. For each library, the test was done with both the full source (including the full big number library) and the minimal source (only

Library	Base	FS	WL	SRC	FlS	WL/FS	WL/FS/SRC	WL/FlS	WL/FS/FlS	WL/FS/FlS/SRC	%
«««< HEAD **Libgcrypt 1.8.2**	66	32	59	54	60	21	26	54	21	21	
mbedTLS 2.9.0	40	40	37	25	39	37	7	32	32	17	
BearSSL 0.5	1	1	1	1	1	1	1	1	1	1	
OpenSSL 1.1.0g											
Reciprocal	32	32	21	24	31	21	13	21	21	13	
Mont.	38	38	29	30	37	29	21	27	27	19	
Mont Const. Time	30	30	21	21	30	21	12	21	21	12	
Mont. Word	31	31	29	16	31	29	14	29	29	14	
======= **Libgcrypt 1.8.2**	64	32	57	54	60	21	26	54	21	21	
mbedTLS 2.9.0	40	35	37	25	39	32	22	39	34	24	
BearSSL 0.5	1	1	1	1	1	1	1	1	1	1	
OpenSSL 1.1.0g											
Reciprocal	32	0	21	24	31	0	0	21	0	0	
Mont.	38	0	29	30	37	0	0	29	0	0	
Mont Const. Time	30	29	21	16	30	20	11	21	29	20	
Mont. Word	31	0	29	21	31	0	0	29	0	0	
»»»> 44b84116d7fce3a7c7528a65139201032ca5d3b2											

Table 1: Number of Warnings based on Features

	Minimal Source				Full Source			
Library	Baseline	Improved	# Branches	KSLOC	Baseline	Improved	# Branches	KSLOC
Libgcrypt 1.8.2	00:24	00:24	246	6.5	17:42	19:37	1938	11.1
BearSSL 0.5	00:21	00:21	141	3.6	00:21	00:27	141	3.6
mbedTLS 2.9.0	00:07	00:07	214	0.5	01:14	01:45	1471	1.7
OpenSSL 1.1.0g	00:27	00:29	498	18.7	11:00	13:11	1880	28.8

Table 2: Baseline and Improved Analysis Run Time (mm:ss)

the modular exponentiation code). The tests show how difference in size of the program affects the running time. The comparison also shows that the increase in processing time in the improved analysis is small even for the larger libraries such as libgcrypt and OpenSSL.

4.3.4 Result Classification. Of all the experiments, two of the tests were classified to understand the category for the positives. The first experiment classified is the baseline experiment. Ideally no high-risk results are eliminated when moving from the baseline to a higher precision result. For most of the libraries the best results were given when combining the full source with the analysis that utilized field-sensitivity and white-listing, so those results were classified to compare against the baseline.

The results from the experiments are sorted into four categories: false-positives, error-handling, 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.

Error-handling results are branches which lead to exit or error handling code. Error-handling 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 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 sensitive data but is considered acceptable. The white-list is often very small yet can eliminate many of the false-positive or low-risk results.

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 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.

4.3.5 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 **??**. 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 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.

4.3.6 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 **??**.

```
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 10: Libgcrypt 1.8.2 - mpi-internal.h lines 113-120

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.

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 figure **??**.

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.

				Low-Risk		High-Risk	
Library	Base	Improved	FP	Error-Handling	Low Leakage	Looped	Controllable
Libgcrypt 1.8.2	64	24	8	10	1	5	0
mbedTLS 2.9.0	40	25	2	13	3	3	1
BearSSL 0.5	1	1	0	1	0	0	0
OpenSSL 1.1.0g							
Reciprocal	32	13	3	2	6	2	0
Montgomery	34	21	3	5	9	2	2
Montgomery Const. Time	30	12	0	7	3	0	2
Montgomery Word	22	14	2	6	5	1	0

Table 3: Result Classifications: Base - Baseline Positives, Improved - WL/FS/SRC Positives

```
1  e = ep[i];
2  e = (e << c) << 1;
3  ...
4  for(;;)
5    if (e == 0)
6      {
7        j += c;
8        if ( --i < 0 )
9          break;
10
11       e = ep[i];
12       c = BITS_PER_MPI_LIMB;
13     } else ...
```

Figure 11: 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. The branch on line 6 (figure **??**) is in the main processing loop, so it is classified as a looped high-risk result.

```
1  count_leading_zeros (c0, e);
2  e = (e << c0);
3  c -= c0;
4  j += c0;
5
6  e0 = (e >> (BITS_PER_MPI_LIMB - W));
7  if (c >= W)
8    c0 = 0;
9  else
10   {
11     if ( --i < 0 ) {
12         e0 = (e >> (BITS_PER_MPI_LIMB - c));
13         j += c - W;
14         goto last_step;
15       }
16     else
17       {
18         c0 = c;
19         e = ep[i];
20         c = BITS_PER_MPI_LIMB;
21         e0 |= (e >> (BITS_PER_MPI_LIMB - (W - c0)));
22       }
23   }
```

Figure 12: Libgcrypt lines 635-658

At the end of each loop iteration, a non-constant time for-loop runs (Fig **??**). 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.

```
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 13: Libgcrypt mpi-pow.c lines 667-695

The while loop shown in figure **??** 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. This result was covered in an attack paper, which showed that knowing the sequence of squares and multiplies could lead to leakages of set bits in the exponent. This attack could function by having a program run on the same machine that is executing the vulnerable code[?]. Another attack is based on this same code, which does not require a program running on the machine, it uses a hardware technique to measure electro-magnetic emanations from laptops[?].

Results like these are important because the attacks are sophisticated, but are based on the knowledge that information about the target information is available in the sense to be measured. Gaining knowledge that this branch is possibly tainted by confidential data does not guarantee a side-channel. This result is helpful to identify code where a side-channel may exist.

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

Figure 14: Libgcrypt mpi-pow.c lines 702-707

4.3.7 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. Although the modulus is a public value, this result shows that branch conditions that lead to paths which take time differences based on execution path. Here there is an untrusted input, the plain-text represented as an integer A compared directly with a value marked sensitive N. Depending on the value of the untrusted data in regards to the sensitive data, the code either does a copy operation or conducts a modulus operation.

```
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 ) );
```

Figure 15: 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 $E \rightarrow p$. 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. This is again a similar implementation of the sliding-window exponentiation in Libgcrypt. Based on the value of the private key exponent, a certain sequence of square operations happens and once a complete window has been processed, the multiply operation occurs. This was resolved differently than libgcrypt, mbedTLS used additional montgomery reductions alongside data blinding[?], but even that was later discovered to be vulnerable[?].

```
while( 1 )
{
    ...
    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 )
    {
        for( i = 0; i < wsize; i++ )
            MBEDTLS_MPI_CHK( mpi_montmul( X, X, N, mm,
                &T ) );

        MBEDTLS_MPI_CHK( mpi_montmul( X, &W[wbits], N,
            mm, &T ) );

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

Figure 16: mbedTLS 2.9.0 - bignum.c lines 1717-1773 (abbreviated)

4.3.8 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

```
1       /*
2        * process the remaining bits
3        */
4       for( i = 0; i < nbits; i++ )
5       {
6           MBEDTLS_MPI_CHK( mpi_montmul( X, X, N, mm, &T
                ) );
7
8           wbits <<= 1;
9
10          if( ( wbits & ( one « wsize ) ) != 0 )
11              MBEDTLS_MPI_CHK( mpi_montmul( X, &W[1], N,
                    mm, &T ) );
12      }
```

Figure 17: mbedTLS 2.9.0 - bignum lines 1775-1786

```
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 18: BearSSL 0.5 - br_i32_tmont.c

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 **??**.

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.

The main loop, shown in figure **??**, still uses the same square and multiply type method, but BearSSL uses the square-and-always-multiply method with a constant time conditional copy to avoid the problem.

```
1       for (k = 0; k < ((uint32_t)elen << 3); k ++) {
2         uint32_t ctl;
3
4         ctl = (e[elen - 1 - (k >> 3)] >> (k & 7)) & 1;
5         br_i32_montymul(t2, x, t1, m, m0i);
6         CCOPY(ctl, x, t2, mlen);
7         br_i32_montymul(t2, t1, t1, m, m0i);
8         memcpy(t1, t2, mlen);
9       }
```

Figure 19: BearSSL 0.5 - br_i32_modpow.c main loop

4.3.9 OpenSSL 1.1.0g. 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 **??**, 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 figure **??**, 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.

```
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 20: OpenSSL 1.1.0g - bn_exp.c lines 250 - 259

Within the same processing loop, the second high-risk looped result can be found, line 6 in figure**??**. The analysis reports this result

```
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 21: OpenSSL 1.1.0g - bn_lib.c lines 741 - 753

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*. In previous versions of OpenSSL, a cache-based attack identified by observing misses that occurred [**?**]. The misses allowed the attacker to know whether the *mod_mul* call was executing from line 18 or line 21. Though this code is reachable, the function should not run if the constant time options are selected for the exponent and modulus. Though this analysis is done on explicit flow, an important implicit flow problem exists with flow from line 6 to *wvalue*. In the montgomery constant time implementation, a defensive scatter gather technique is used to mitigate cache-attacks that identify which cache block was accessed by the index *wvalue* [**? ?**].

```
1  for(;;){
2    ...
3          for (i = 1; i < window; i++) {
4              if (wstart - i < 0)
5                  break;
6              if (BN_is_bit_set(p, wstart - i)) {
7                  wvalue <<= (i - wend);
8                  wvalue |= 1;
9                  wend = i;
10             }
11         }
12
13         /* wend is the size of the current window */
14         j = wend + 1;
15         /* add the 'bytes above' */
16         if (!start)
17             for (i = 0; i < j; i++) {
18                 if (!BN_mod_mul_reciprocal(r, r, r, &
                       recp, ctx))
19                     goto err;
20             }
21         if (!BN_mod_mul_reciprocal(r, r, val[wvalue >>
             1], &recp, ctx))
22             goto err;
23   ...
24 }
```

Figure 22: 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.

Line 1 in figure **??** 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.

```
1  if (a->neg || BN_ucmp(a, m) >= 0) {
2      if (!BN_nnmod(val[0], a, m, ctx))
3          goto err;
4      aa = val[0];
5  } else
6      aa = a;
7
8  ...
9  if (!BN_to_montgomery(val[0], aa, mont, ctx))
10     goto err;                /* 1 */
```

Figure 23: OpenSSL 1.1.0g - bn_exp.c lines 363-368

Line 2 of figure **??** 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 figure **??**. The length of time to execute these branches is longer for the case where *a* is greater than 0 and *m*.

```
1  if (a->neg || BN_ucmp(a, m) >= 0) {
2      if (!BN_mod(&am, a, m, ctx))
3          goto err;
4      if (!BN_to_montgomery(&am, &am, mont, ctx))
5          goto err;
6  } else if (!BN_to_montgomery(&am, a, mont, ctx))
7      goto err;
```

Figure 24: 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 precomputed table of powers.

It is possible that there are true positives within the source code given in figure **??**. This code defines the *bn_gather5* function, but

```
for (wvalue = 0, i = bits % 5; i >= 0; i--, bits--)
    wvalue = (wvalue << 1) + BN_is_bit_set(p, bits);
bn_gather5(tmp.d, top, powerbuf, wvalue);
```

Figure 25: 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 26: OpenSSL 1.1.0g - bn_exp.c lines 1200-1214

```
        j = wstart;
        wvalue = 1;
        wend = 0;
        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(r, r, r, m, ctx))
                    goto err;
            }

        /* wvalue will be an odd number < 2^window */
        if (!BN_mod_mul(r, r, val[wvalue >> 1], m, ctx
            ))
            goto err;
```

Figure 27: OpenSSL 1.1.0g - bn_exp.c lines 1326 - 1350

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 **??**.

Simple Method

The sliding window attacks from the other libraries were also exploitable in earlier version of OpenSSL. The code for the function *bn_mod_exp_simple* is specified as an entry point, although it is not reachable from the main entry point of the algorithm. Line 7 in figure **??** is reported because the branch is dependent on the current bits value of the exponent *p*. This result is directly computed from the exponent and within a loop so the amount of leakage is high if the branch outcome is known. The timing-channels within this implementation was found in the timing difference between the *mod_mul* where *r* is squared [**?**]. The implementation of the *mod_mul* function calls *bn_sqr* if both inputs are the same, otherwise the more expensive *bn_mul* is called.

4.4 Verifying a 2-safety Property

In this section, we discuss another approach to verify constant time code, and compare the precision and accuracy between it and CtChecker. We consider a recent effort that verifies a 2-property, expressing the security of a program as a set of logical statements or conditions, and verifying them automatically using a theorem solver. Our goal was to perform a reasonable comparison that focuses on the advantages of each method, and not the engineering details associated with implementing them.

4.4.1 Background. The paper of Almeida et al. uses an approach based on a reduction of the security of a program *P* to the assertion-safety of a program *Q*, and implements it in a prototype, Ct-verif. [**?**]. The reduction is inspired by prior work on self-composition and product programs. A product program *Q* verifies the security of *P* by simulating two executions in lockstep. It asserts the equality of each public input with its renamed copy at each branching instruction. Public outputs are not taken into consideration, and the technique constructs an output-insensitive product program. Theoretically, the approach is sound and complete in that all safe programs have a correct security verdict, and all unsafe programs have a correct insecurity verdict.

Ct-verif verifies optimized LLVM by implementing their reduction technique. The SMACK verification tool is used as a front-end to compile the annotated C-code via Clang and the resulting compiled and optimized LLVM code is translated to Boogie code. The reduction is performed on the Boogie code and then applied to the Boogie verifier, which uses an SMT logic solver. Ct-verif provides an annotation interface for public inputs.

The implementation does not provide results for the entire source. During its product program verification, if an error state is reached (a violation is found), no transition is enabled and the system will stop its execution. Constant time violations that occurred first in sequence were reported, but identification of violations that occurred afterwards was not guaranteed. To identify all violations, we replaced each error with some constant time code and re-ran the tool on the revised code.

We perform a reasonable comparison between the two tools to better understand the distinct advantages of each method. We aim to avoid the drawbacks of the tools caused by engineering issues; we assume the best setting for each tool.

4.4.2 Limitations of Ct-verif. Although the approach is theoretically sound and complete, the results of a practical interpretation, must be carefully analyzed. To perform a reasonable comparison, we avoid key limitations not present in CtChecker by revising the analyzed source to accommodate Ct-verif's technical flaws. For instance, loops and non-static length arrays are not supported.

	Ct-verif				CtChecker			
Library	Baseline	No Loops	Removed	Diff	Baseline	No Loops	Removed	Diff
Libgcrypt 1.8.2	45	25	0	20	25	25	1	0
BearSSL 0.5	14	10	0	4	10	10	0	0
mbedTLS 2.9.0	100	80	0	20	77	77	1	0
OpenSSL 1.1.0g								
Reciprocal	23	19	0	4	19	19	0	0
Mont.	29	23	0	6	21	21	0	0
Mont Const. Time	39	28	0	11	25	25	2	0
Mont. Word	16	15	0	1	15	15	0	0

Table 4: The baseline file (version 2) accommodated the excluded source. Positives caused by loops were removed from the baseline in a separate file (version 3). A new file removed the remaining positives (version 4).

Loop invariants are automatically computed to verify program loops. When counters are used to index arrays, a proof of security would require Ct-verif to infer that the memory accesses are in the range of public values. ~~Although there are options to avoid this difficulty, such as enforcing a loop limit or unrolling a loop, the [actually, why doesn't this work?].~~ To avoid this limitation, loops were replaced with a branch statement that depended on the values that affected the loop bounds.

Constant sized arrays are supported by adding an extra annotation, but variable sized arrays are not. To ensure memory accesses are not in the range of secret values, array indexing needs to be fixed. We dictated that any array access was fixed in between these annotated bounds. False positives caused by loops or variable array accesses were included in a separate result for Ct-verif. The revisions made to replace a for-loop and a variable array access in *MPN_NORMALIZE* are shown in figure 30.

We did not include the full source in all of the libraries, except for in BearSSL, which has a small amount of reachable code from the modular exponentiation function. Undefined values, such as those that are assigned to the return value of a function and are excluded from the analysis, were replaced with new variables. CtChecker handles excluded source code by conservatively tainting the return value of a function whose parameter is tainted. In contrast, Ct-verif taints the return value regardless of the state of the parameters. In this direct comparison, we replaced the excluded function's return value with some new variable. If the return value was part of a branch condition, and the function was called with tainted parameters, this line was counted as a positive for both tools. In figure 28, *gcrympih_lshift* is not linked in the analysis, so its return value is unknown. Thus, *carry_limb* is undefined before the branch, and the verifier throws an error. The revision occurs on line 2 wherein *carry_limb* is given a reserved public input.

4.4.3 Results. We note that all true positives were reported among both tools, so we observed comparable accuracy. Mainly, the results explain a couple of reasons for the significant difference between the precision observed. Several versions of the cryptography algorithms were created before comparing the precision of the two tools against one another. There are three versions (version 2, version 3 and version 4, where version 1 is the original file), each of which was developed from the version number before it.

```
1 carry_limb = _gcry_mpih_lshift( res->d, rp, rsize,
      mod_shift_cnt);
2 carry_limb = PUBLIC_VALUE;
3 rp = res->d;
4 if ( carry_limb )
5 {
6     rp[rsize] = carry_limb;
7     rsize++;
8 }
```

Figure 28: Libgcrypt 1.8.2 - mpi-pow.c lines 717-723

First, a baseline version was created to accommodate minor limitations or differences between the tools, such as the way in which they operate with some undefined values. Any assignment statement with excluded source was rewritten to assign the variable to a public or private input, depending on the function return value. If any of the function's parameters were tainted, we conservatively tainted its return value. Thus, a function call inside of a branch condition was counted as a positive when any of its parameters were tainted. These positives were counted the same for each tool, and were only observed in OpenSSL.

The second version was constructed from the first, and it allowed us to examine the number of positives introduced from computing loop invariants. Each erroneous loop was replaced, and each variable array access was changed to a fixed one. Loops were replaced by a model that leaked the same information as the loop: a *if (e) then* statement where *e* contains the loop bounds. Figure 30 is an example of two revisions in Libgcrypt, one for a loop and one for an array access. Ct-verif supports an annotation for a static number of values to be marked as public. Variable length arrays are not supported, so variable array element accesses needed to be removed. Every array index was fixed within the bounds of the array.

The difference in the number of positives reported by Ct-verif in the baseline and no loops versions is the number of positives caused by computing loop invariants only. The number is a significant portion of the total positives reported on each version. As expected,

CtChecker reported the same number of positives after removing loops.

The third version was created from the second to remove each erroneous line reported by Ct-verif, resulting in a subset of the original file that was free of any constant time violation. This final version allowed us to evaluate the precision of CtChecker in another way; any positive reported on this version was exclusive to CtChecker, and was expected to be a false one.

Ct-verif is flow sensitive; the order of statements in a program may affect the analysis. CtChecker's flow insensitivity is one cause of false positives in the results.

Version 4 was constructed by replacing non-constant time code to intentionally observe 0 positives by Ct-verif. The positives reported by CtChecker in this final version was the result two design choices distinct from Ct-verif. First, CtChecker is a flow insensitive analysis (the order of statements in a program does not matter). In figure 29, variable *i* is not tainted until line 851, but flow insensitivity causes CtChecker to flag line 1041 since *i* is assigned to a tainted value, *bits*. Second, rather than tainting a range of values, CtChecker taints the variable pointing to them. In figure 32, *ep* is a pointer to sensitive data.

Overall, CtChecker exhibited a large improvement precision over Ct-verif, a result apparent in the difference between the number of positives reported by each tool in version 2. CtChecker showed equal or improved precision over Ct-verif when disregarding significant limitations, such as loop invariants. This result is apparent in the difference between the number of positives reported by each tool in version 3.

```
1  tmp.d[0] = (0 - m->d[0]) & BN_MASK2;
2  for (i = 1; i < top; i++)
3      tmp.d[i] = (~m->d[i]) & BN_MASK2;
4  tmp.top = top;
5  ...
6  for (wvalue = 0, i = bits % window; i >= 0; i--, bits
       --)
```

Figure 29: OpenSSL 1.1.0g - bn_exp.c lines 741-1041

4.4.4 Conclusion. Perhaps the simple heuristic used for-loop invariant generation is the primary reason the 2-property analysis saw a lesser precision. Additionally, while deductive verification is sufficient for verifying non-trivial programs, like the ones in this paper, it remains difficult to use. As previously mentioned, there is no trivial method to interpret verification failures. CtChecker uses many fewer annotations to run its analysis, and provides a complete list of violations for each source line.

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

Figure 30: Libgcrypt 1.8.2 - mpi-internal.h lines 113-120

```
1  if ( rp == ep )
2  {
3      /* RES and EXPO are identical.  Allocate temp.
           space for EXPO.  */
4      ep_nlimbs = esec? esize:0;
5      ep = ep_marker = mpi_alloc_limb_space( esize, esec
           );
6      MPN_COPY(ep, rp, esize);
7  }
```

Figure 31: mbedTLS 2.9.0 - bignum.c lines 1778-1786

5 RELATED WORK

5.1 Detecting Timing Channels

There are several approaches to detecting non-constant time code using both static and dynamic taint tracking. In addition to entirely software based approaches, hardware language approaches have been used to target leakage from low-level hardware features. Efforts have been made to eliminate the need for these analyses by creating languages that restrict the amount of leakage through covert channels. There are also fuzzing methods which are able to detect non-constant time implementations[**?**].

There are static analysis tools which will verify that a source is constant time given a set of security annotations, such as VirtualCert[**?**]. VirtualCert provides a static analysis which operates a flow-insensitive type analysis. VirtualCert is used for verifying isolation for virtualization systems. Our analysis works in a similar way but avoids the need for additional annotations.

Almeida et al. provided a way to verify programs using deductive verification [**?**]. These analyses accept or reject programs on the basis of being constant-time. In Almeida's work, they have a similar analysis which uses the same points-to analysis used in this work, to track memory operations. Their work is based on self-composition, which is considered both sound and complete, however the analysis is not as they specify sources of incompleteness. Their analysis does not provide a field-sensitive analysis without additional annotations as ours does.

Flow Tracker is another static analysis that looks for non-constant time implementations due to implicit flow[**?**]. FlowTracker is a flow-sensitive analysis that operates on LLVM IR code and requires additional annotations. The analysis presented in this paper does not require additional annotations to achieve a field-sensitive implicit flow analysis.

CaSym is a static analysis which looks to find cache-based side channels by processing LLVM IR[**?**]. Brotzman et al. propose different cache models to achieve results which are architecture independent and explores all execution paths. Their analysis achieves their results by assuming a cache model and uses symbolic execution to determine the results. Our analysis does not require a cache model and, instead of symbolic execution, uses information flow. Since we do not assume a cache model, our analysis may miss positives that require details of the cache to observe.

CacheD is a trace-based analysis which identifies cache-based timing-channels using taint tracking and symbolic execution[**?**]. The strength of CacheD is its ability to identify the location of the vulnerability. CacheD does look for differences in the state of the cache, which are not accounted for in this work. The analysis in

Library	Baseline	No Loops	Removed
Libgcrypt 1.8.2	23	18	0
BearSSL 0.5	16	3	0
mbedTLS 2.9.0	84	67	0
OpenSSL 1.1.0g			
Reciprocal	14	14	0
Mont.	19	16	0
Mont Const. Time	34	25	0
Mont. Word	16	15	0

Table 5

this paper also identifies the locations of vulnerabilities for timing channels, but does so statically.

FaCT takes a whole different approach to trying to eliminate non-constant time methods[**?**]. FaCT is a programming language proposed specifically to help developers generate assembly code that is free of timing channels. They argue that using C does not help developers safely handle sensitive data. This approach requires libraries to be rebuilt in this language. Since many libraries make use of C currently, this analysis and analyses previously mentioned are still necessary to identify possible timing channels in deployed software.

Zhang et al. propose software-hardware co-design to address the issue of cache-based side channels[**?**]. In their work, they show that it is possible with a small overhead to enforce control over information flow at a hardware level specified in software. Their solution involves masking time variations by having a consistent timing regardless of actual execution time. Our analysis is attempting to find the code which causes the timing variations.

Fuzzing methods target a somewhat different area than the analysis proposed in this paper. Fuzzing methods commonly look to exploit attacks using invalid inputs or erroneous execution paths [**?**]. Cryptosystem vulnerabilities are often not a consequence of exception but one from monitoring execution times and control-flow choices. Fuzzing methods may miss results found in other types of analyses targeted at identifying timing channels.

5.2 Known Attack on Cryptosystems

The high-risk results identified confirm attacks found in previous papers in Libgcrypt and OpenSSL. Kocher showed that various cryptography algorithms could be compromised through timing attacks[**?**]. The modular exponentiation functions were implemented using a sliding window method in early versions of the libraries.

It has been shown, using Libgcrypt as an example, that the ability to distinguish a transition from a sequence of squares to a multiply execution can leak a majority of the secret key. [**?**]. This attack exploited the Montgomery reduction algorithm and followed each multiplication to mount an amplification attack. Though the attack is a cache-based attack, the length of the for-loop is not constant time and this result is reported among the high-risk findings in this paper.

OpenSSL has an attack very similar to the Libgcrypt square-and-multiply attack, as shown by Percival[**?**]. This showed that cache misses could be used to identify the multiplications in the square-and-multiply sequences. This vulnerability is a result of non-constant amount of squares followed by a multiply, a result reported in our findings.

OpenSSL had a different attack paper by Brumley and Boneh[**?**]. This paper also showed that timing attacks do not need to be mounted from the same machine. This is dangerous because these attacks were able to be mounted remotely. Another dangerous find is that even with virtual machines used for isolation, the secret key could be extracted using the methods they describe. This attack was based on the timing difference between comparisons of the current value of the cipher text to one of the prime numbers used to compute the modulus. The actual source for this timing attack was in a different file than the one analyzed for the results in our paper. The same reduction timing attack was found in mbedTLS, shown by Dugaurdin et al [**? ?**].

Many of these vulnerabilities lie at the source/IR code level, but the attacks need more information to be successful. Hence, it is still valuable to know which branches could potentially lead to an exposed attack surface.

6 CONCLUSION

In this work, the addition of several features to a baseline information-flow analysis was tested. On its own, field-sensitivity is usually not adequate to reduce the number of results. Adding additional source code improved the precision of the analysis since field-sensitivity is heavily reliant on the points-to analysis for type information and tracking. The addition of a whitelist to eliminate the results deemed safe by examination also helps reduce the list to only positives which are not expected.

Tests show that and the effectiveness of field-sensitivity in reducing the number of positives depends on a number of factors. The field-sensitive results worked best when adding additional source code. Due to the reliance on the points-to analysis to provide information for type and offset information, the best results were achieved when enough source was provided so that sensitive fields were accessed within the source code.

This analysis, given only a list of sensitive data, produces a list of source lines which have conditions based on sensitive data. The

analysis produces fewer results when compared to the baseline, and many of the high-risk results are confirmed from previously published attack papers. The libraries that did see a reduction in results saw a 40-60% reduction in results. Using the classification methods specified, the number of results can be sorted and prioritized to potentially identify non-constant time branches.

Future Work

Currently, the classification of the results is done manually, but the criteria for the categories are simple. Ideally, an addition to the system would automatically classify the results into at least FP, validation, low, and high-risk results. This way developers know which results to understand first, and then decide whether a positive needs to be addressed or not.

Additionally, field sensitivity is heavily dependent on the points-to analysis used. An understanding of the points-to analysis limitations aids in interpreting results. For example, if the points-to analysis used in this paper found that type information across function contexts did not line up, then the node would have no type-information, meaning all field sensitivity is lost for that node.

Received February 2019