Bachelor thesis
Computing Science

Radboud University

An exploration of static analysis used on C cryptography libraries

Author:
Auke Zeilstra
s4751701

First supervisor/assessor:
dr. P. Schwabe (Peter)
p.schwabe@cs.ru.nl

Second assessor:
dr. ir. E. Poll (Erik)
e.poll@cs.ru.nl

January 19, 2020

Abstract

Cryptography libraries are commonly written in the programming language C. These libraries feature unusual code, such as big-integer arithmetic and implementations for cryptographic primitives. C contains subtleties that even people familiar with the standard struggle with [24]. This study aims to determine to what extend static analysis tools for C are helpful in the process of finding bugs in C cryptography libraries.
Secondly, the TweetNaCl library contains a bug resulting from a type mismatch inside a for-loop condition. Compiling the library with GCC does not generate a warning, even with all warnings enabled. An attempt will be made to reliably detect this bug using data-flow analysis.

Three popular static analyzers were used to perform static analysis on the source code of four cryptography libraries. The statically analyzed libraries are BearSSL, mbed TLS, Libsodium and TweetNaCl. The reports were checked for correctness and reasoning behind their occurrence. Common warnings from two static analyzers are summarized and a couple of true positives are examined. Lastly, coding practices specific to cryptography that lead to the generated reports are summarized.
The TweetNaCl bug is characterized and a strategy is presented to catch the bug using data-flow analysis. A prototype implementation called Sans_t is presented in the form of a GCC plugin, which is used on the four cryptography libraries from the other half of this study.

The results indicate that static analysis can be used to find bugs in C cryptography libraries, without cryptography specific code practices causing an infeasible amount of false positives. Based on the presence of two potentially critical bugs, statically analyzing a cryptography library before a release is recommended.
The Sans_t prototype detects the TweetNaCl bug, at the cost of generating false positives for the other libraries due to not performing value range analysis.

Contents

Chapter 1

Introduction

1.1 C static analysis

Cryptography libraries aim to provide a simple interface to achieve secure communication. These libraries are commonly written in the programming language C. Examples include OpenSSL, NaCl, and cryptlib. The ANSI C language standard has been around for more than 30 years, and the language remains useful for its efficiency and portability. At the same time, C is a language that provides a lot of pitfalls. It does not provide classes such as `BigInteger` like Java, nor does it present the memory safety of Rust, which is similar in terms of syntax. A cryptography library written in C requires extensive checking and testing of its code, by its developers and the outside world. Static analyzer are used to analyze code without executing it, helping users by pointing out alarming code constructions. After the Heartbleed bug [8] in OpenSSL was publicly disclosed in 2014, few static analyzers were found that were able to catch the bug [38]. This presents the question whether modern static analyzers such as Facebook Infer and the Clang Static Analyzer, are helpful for finding bugs in present-day cryptography libraries.
If results show a feasible amount of false positives, paired with a few true positives, authors of cryptography libraries might be inclined to use such tools whenever a release candidate is developed. This could lead to fewer bugs in actual releases, and possibly a coding style adapted to prevent false positives produced by the employed static analyzer.
This study will apply static analyzers designed for regular C code to cryptography libraries. The approach will be focused on the code that handles the cryptographic primitives. This contrasts with similar studies, which either focus on protocol implementations [9] or actual verification of a cryptographic primitive [2].

1.2 TweetNaCl type mismatch bug

One of the cryptography libraries that is statically analyzed is TweetNaCl. TweetNaCl contains a bug resulting from a type mismatch inside a for-loop condition. The condition will compare a 32-bit signed integer to a 64-bit unsigned integer. The latter is typically a parameter describing the size of an array, which is passed by reference.
In TweetNaCl, the construction leading to the bug is not warned about during compilation using GCC with the `-Wall` flag. The `-Wsign-compare` flag, included in `-Wextra`, warns about comparisons between integer expressions with a different signedness. Thus, the bug is partially exposed by GCC's extra warnings flag. The bug may result in undefined behavior when TweetNaCl usage guidelines are ignored. Figuring out a way to detect the presence of the bug would help avoid it in the future.
To catch the bug using a GCC plugin, data-flow analysis is performed to trace which variables are influenced by long unsigned parameters. Doing so requires correctly interpreting C constructions such as `goto` expressions and conditional expressions. Basic blocks are created, each with their own variable influence map. These maps are used to propagate flags for variables, either to the start of successor blocks or right before an (array) reference is used. The references can be checked for the bug using the propagated flags. The bug is not a common occurrence, nor is it heavily researched. There are static analysis checkers with the ability to catch part of the bug, such as Splint [13]. Splint reports potentially harmful comparisons, but this check is not specific enough to catch the bug without unnecessary false positives.

Chapter 2

Preliminaries

2.1 Static analysis tools

Static analysis is concerned with the analysis of a program without actually running it, usually by automated means. Generally, this process consists of 3 phases, as described in [11]:

1. Observe the program instructions. A suitable level of abstraction must be chosen for the problem at hand. For example, if all language information is important, one could choose Abstract Syntax Trees (ASTs).
2. Build a model of the program state, using appropriate abstractions.
3. Run the model over (a set of) abstract states.

There are a variety of ways to build the model. The following list sums these up for the static analyzers that will be used for analyzing the cryptography libraries:

- Facebook Infer [6] uses automatic verification with separation logic [30], an extension to Hoare logic [16]. Separation logic includes support for pointer data structures and transfer of ownership. An important concept in Infer is that, whenever a procedure allocates a new object, it should either deallocate this object itself, or make it available to the caller. Specification synthesizing is another important feature of Infer, steering away from interactive proofs. Hoare triples are generated bottom-up, roughly meaning that a composite procedure will 'inherit' deduced specifications from the procedures it calls. Infer is used in order to verify memory safety in C code.
- The Clang Static Analyzer [22] builds a control-flow graph, on which it performs path- and flow-sensitive analysis. These techniques are used to reason about variable values and their propagation. Flow-sensitive analysis takes into account the order of statements, whilst

path-sensitive analysis takes into account conditions in branch guards. The main challenges of this approach include avoiding a state explosion and choosing the right model precision whilst keeping false positive rates in mind.

- Splint [13] uses similar techniques to the above; procedure annotations for pre- and postconditions, flow-sensitive analysis and resolving preconditions from earlier statements. However, Splint is based on several compromises, such as merging paths during flow-sensitive analysis and using loop heuristics as a simplification. A clear trade-off was made between performance and false positive rate.

An honorable mention is CBMC [23], a bounded model checker for both user-defined- and automatically inserted assertions in C code. CBMC performs symbolic execution on an intermediate representation, which translates conditional statements and loops to guarded goto statements. Loop unwinding is done up to a certain depth and Static Single Assignment (SSA) form is used. The program is represented as a single equation, which is transformed into a CNF formula. A SAT solver is used to find counterexamples to any of the assertions. Due to compilation problems, CBMC will not be included in the report.

2.2 GCC

The GNU Compiler Collection (GCC) supports various programming languages, of which C was the first one. GCC and Clang are the mainstream options when compiling C code for Linux related projects.
When lowering source code from a high-level language, the compilation process of GCC generally uses three Internal Representations (IR), one for each lowering stage [36]. In order, these are GENERIC, GIMPLE, and RTL. GENERIC and GIMPLE are machine-independent representations and therefore the preferred analysis target for static analysis of source code targeting multiple platforms.
GCC skips AST representation for C files. The front-end directly produces GENERIC IR, in which functions are represented as trees; implementation details are found in `tree.def` [15].
Since the GENERIC IR is used to lower name spaces, including scopes, this is the representation for analyzing the code as close to its origin as possible. This strategy is used in IDE's on the AST, because the representation still captures high-level abstractions that would be compiled away during the translation to intermediate code [33]. Because ASTs are skipped, procedural lowering has already happened in GENERIC. Earlier mentioned static analyzers favor procedural analysis. The bug in question is associated with procedures and their usage of array-length parameters, meaning the lower-

ing level of GENERIC is suitable for catching the bug.
The GCC plugin system allows users to register callbacks for specific passes in the compilation phase. The callback is registered in a C/C++ project. The project is build into a shared object file, so that it can be included whenever invoking gcc using the `-fplugin=...` flag, without needing to re-compile gcc. Particularly helpful in this case is the plugin event `PLUGIN_FINISH_PARSE_FUNCTION`. Whenever the front end finishes parsing a function into GENERIC, the callback receives the `function_decl` tree as event data. The tree type is defined as a pointer to a `tree_node`. Trees have type codes, associated with language-independent, but also language-dependent constructs in case of C/C++ [35].

For this project, the latest version of GCC at the time of development will be used, which is GCC 9.2.0.

Chapter 3

Type mismatch bug

In this chapter, a type mismatch bug that appears in TweetNaCl is characterized and examples of the bug are shown. A strategy aiming to catch the bug is presented, along with a GCC plugin called Sans_t, which implements this strategy.

3.1 Bug characterization

The usual environment for the bug to take place is any function with a parameter field intended to be used to specify the size of some array or indirect reference. The type of the size parameter is normally `unsigned long` or `unsigned long long`, both typically with a 64-bit precision. The array can be passed to the function or declared inside. Within this function, a for-loop uses a short loop counter variable `i`, for example a signed integer with a precision of 32 bits.
The initialization value of the loop-counter variable can be any integer, but is typically `0`. A conditional expression compares the short loop counter against a long unsigned integer `data_len`. This can be the size parameter or a long integer derived from the size parameter. The long unsigned integer has a higher precision and represents the size of some array `data`. For the example, we will assume a precision of 64 bits. The comparison is of the form:

$$\textit{short integer}\ \mathrm{i} < \textit{long integer}\ \mathrm{n},$$

with the relational comparison optionally being '<='.
The afterthought consists of a simple increment, typically a pre/post-increment of the loop counter variable. Inside the for-loop body, the loop counter variable is used as an index for the array. This setup allows for buffer overreads whenever the array size, the long integer, is bigger than the maximum value of the loop counter variable. A similar situation can take place whenever one accesses the array backwards, using a decrement

afterthought and a '>' or '>=' relational operator.
An instance of the bug is shown in Figure 3.1.

```
1 #include <stddef.h>
2 void all_zero(unsigned char *data, size_t data_len){
3     signed int i;
4     for(i = 0; i < data_len; ++i)
5         data[i] = 0;
6 }
```

Figure 3.1: A simple instance of the bug.

3.2 Reason and variants

The reason that the bug occurs is the range difference between the compared integers. The behavior this causes is dependent on the value of the long integer. Note that we assume the usage of two's complement for signed number representation. In Figure 3.2 on the next page, we can distinguish three cases if `len` were assigned a different value on line 4. Note that the behavior in the cases is not specific to the types used in Figure 3.2; this is just an example of a type combination leading to the bug.

1. $0 <= len <= SHRT\_MAX$:
   `len` is in the range of the signed short. For the comparison `i < len`, `i` is promoted to an `unsigned int`. Since `i` stays within the bounds of `len`, the promotion simply means the `i` is represented wider during comparison.

2. $SHRT\_MAX < len <= UINT\_MAX - SHRT\_MAX$:
   The counter `i` will increment until it reaches `SHRT_MAX`. The iteration after this, the pre-increment causes it to loop around to its most negative value, `SHRT_MIN` $= -2^{15}$. The following representations explain what happens in the comparison:

Decimal:	-2^{15}
Unsigned short:	$10000000\ 00000000_{\text{b}}$
(Un)signed int:	$11111111\ 11111111\ 10000000\ 00000000_{\text{b}}$

   Alternatively, $-2^{15} \equiv 2^{32} - 2^{15} \equiv (2^{32} - 1) - (2^{15} - 1) \pmod{2^{32}}$. Thus, due to promotion rules, the `unsigned int` representation will be compared to `len`. Clearly $i = 2^{31} + 2^{30} + ... + 2^{15} > len$, causing the for-loop to stop immediately.

3. $UINT\_MAX - SHRT\_MAX < len <= UINT\_MAX$:
   This is the situation depicted in Figure 3.2, in which the loop body is executed a single time with a negative `i` value. Because a less-than operator is used, the loop will end at its latest when
   $len = UINT\_MAX = 2^{32} - 1$.
   For this value of `len`, the counter `i` takes on the values
   $\{0, 1, ..., SHRT\_MAX, SHRT\_MIN, SHRT\_MIN + 1, ..., -2\}$ inside the for-loop body. Because $-1 \equiv 2^{32} - 1 \pmod{2^{32}}$, the loop ends when $i = -1$.

The output of the code in Figure 3.2 is shown in Figure 3.3.

Finally, there is another variant of the bug, which occurs when both the loop-counter variable and the length variable are of a type shorter than `int`. This can be seen in Figure 3.4 on the next page, along with its output in Figure 3.5 (also on the next page). The ranges of `unsigned short len` and `signed char i` are within `signed int` range. Therefore, any length over `CHAR_MAX` causes an infinite loop if the `~i` condition was not present. Note that memory preceding `(arr+len)` is accessed.

```c
#include <limits.h>
#include <stdio.h>
int main(int argc, char *argv[]){
    unsigned int len = (UINT_MAX - SHRT_MAX) + 1;
    signed short i;
    printf("%6s\n", "i");
    for(i = 0; i < len; ++i)
        if(!(i&((1 << 13)-1))) printf("%6d\n", i);
    return 0;
}
```

Figure 3.2: The last iteration of the loop uses a negative i.

```
     i
     0
  8192
 16384
 24576
-32768
```

Figure 3.3: The output produced by the code in Figure 3.2.

```c
#include <limits.h>
#include <stdio.h>
int main(int argc, char *argv[]){
    unsigned short len = CHAR_MAX + 1;
    unsigned short arr[len<<1];
    for(unsigned short i = 0; i < (len << 1); ++i)
        arr[i] = i;
    signed char i;
    printf("%4s    %s\n", "i", "(arr+len)[i]");
    for(i = 0; i < len && ~i; ++i)
        if(!(i&15)) printf("%4d    %12u\n", i, (arr+len)[i]);
    printf("Loop ended at %d to prevent infinite loop.\n", i);
    return 0;
}
```

Figure 3.4: Promotion of both comparison operands to integer.

```
                   i    (arr+len)[i]
                   0             128
                  16             144
                  32             160
                  48             176
                  64             192
                  80             208
                  96             224
                 112             240
                -128               0
                -112              16
                 -96              32
                 -80              48
                 -64              64
                 -48              80
                 -32              96
                 -16             112
Loop ended at -1 to prevent infinite loop.
```

Figure 3.5: The output produced by the code in Figure 3.4.

3.3 TweetNaCl example

The following definitions from TweetNaCl are relevant to the code that contains the type mismatch bug:

1. `#define FOR(i,n) for (i = 0;i < n;++i)`

2. `typedef unsigned long long u64`

Figure 3.6 on the next page shows an instance of the bug in the cryptography library TweetNaCl [4]. The function verifies the signature of the signed message `sm` of length `m`; `sm` is an array of bytes. The public key used to verify the signature is stored in `pk`. If the signature is valid (check in line 24), the message `m` and its length `mlen = n - 64` are stored and the function returns `0`, signaling that the content of the message is authenticated. The *minimum* ranges of the variables in the code snippet, as stated in the C17 ballot [19] and C90 [17]:

$$\begin{array}{rcl} \text{Loop counter variable:} & \texttt{int i} & \in \quad [-2^{15}, 2^{15}\text{–}1] \\ \text{Array size:} & \texttt{u64 n} & \in \quad [0, 2^{64}\text{–}1] \end{array}$$

In the expanded `FOR`-macro in line 13, the less-than comparison `i < n` in combination with the pre-increment expression `++i` may cause at least the following two problems:

1. The short integer reaches its maximum value and wraps around to its minimum value, which could still result in the condition evaluating to false depending on the value of `n`.

2. After wrapping around, a signed short integer such as `i` in the above example, will represent a negative value. A negative index `i` for an array of type `T` results in memory access of `sizeof(T)` bytes located at `array_base_address - sizeof(T)*i`. This is undefined behavior.

In practice, two details should be noted:

1. The ranges are almost always problematic, unless really exotic hardware is used. Even if unsigned long integers use their minimum range, and the integer uses the minimum signed long range $[-2^{31}, 2^{31}\text{–}1]$, there are still 2^{31} unsupported array sizes.

2. On common platforms, signed integers use the $[-2^{31}, 2^{31}\text{–}1]$ range. This means that for the unwanted/undefined behavior to occur in TweetNaCl on these platforms, one would be dealing with a message of at least 2^{31} bytes. NaCl's *Validation and verification* [26] section suggests users to "put a small global limit on packet length", hinting at 4096 bytes or less, the maximum test size.

```
1  int crypto_sign_open(u8 *m,u64 *mlen,const u8 *sm,
2                       u64 n,const u8 *pk)
3  {
4      int i;
5      u8 t[32],h[64];
6      gf p[4],q[4];
7
8      *mlen = -1;
9      if (n < 64) return -1;
10
11     if (unpackneg(q,pk)) return -1;
12
13     FOR(i,n) m[i] = sm[i];
14     FOR(i,32) m[i+32] = pk[i];
15     crypto_hash(h,m,n);
16     reduce(h);
17     scalarmult(p,q,h);
18
19     scalarbase(q,sm + 32);
20     add(p,q);
21     pack(t,p);
22
23     n -= 64;
24     if (crypto_verify_32(sm, t)) {
25         FOR(i,n) m[i] = 0;
26         return -1;
27     }
28
29     FOR(i,n) m[i] = sm[i + 64];
30     *mlen = n;
31     return 0;
32 }
```

Figure 3.6: The crypto_sign_open function in TweetNaCl version 20140427.

3.4 Bug-catching strategy

The following strategy is used to search for the bug in the GCC plugin Sans_t:

0. Receiving the GENERIC function tree(s).
   Receive the GENERIC function representations from GCC. Only the procedural abstraction has happened yet, meaning we can reason about name spaces, data and control.

1. Parsing the GENERIC function tree(s).
   Walk the GENERIC function tree and produce basic blocks, a convenient representation for analysis. The conditional-, goto- and label expressions influence the structure of the basic block graph, which takes on the shape of a directed, cyclic graph. The blocks are influenced by scoping and the statements contained in the source code. Scopes are described by bind expressions, each containing a chain of active variables that are introduced within its body, and a body containing statements. Statements are grouped inside a statement list node unless only a single one is included.
   Two statements that are of particular interest are the ones describing how variables influence each other, the declaration- and modification expressions. Whenever a variable is flagged for having a specific property, the property may transfer to other variables inside one of these expressions. For the class of bugs we aim at detecting, there are two 'stages' of flagged variables. The first stage consist of variables influenced by a long integer parameter. The second stage consist of variables compared to first stage flagged variables in a conditional expression comparison with a precision difference.
   The challenge is to save each of these nodes in such a way that we can symbolically execute the source code, without calculating any values. All paths need to be considered and nodes like the bind expression must be used to keep track of the active variables for each of these paths.

2. The first flagging stage.
   Any long integer parameter declaration is initially flagged. Then, for every block available, calculate the effect of the block on all of the flagged variable states going into the block. This results in another set of flagged variables. Control flow needs to be taken into account; due to the problem involving a cyclic graph, we will need to revisit certain nodes whenever a new flagged variable state is added by a back edge. An example would be a `goto` expression jumping back to a label expression, with statements (un)flagging variables in between.

3. The second flagging stage.
   Variables with a lower precision than the ones they are being compared to should also be flagged, but their flagging reason is different. The following scenario needs to cause a second-stage flag:
   Assume integer variable y_1 to be first-stage flagged, and variables $x_1, \ldots, x_n$ to be on the other side of integer variable y_1 in a relational

operator inside a conditional expression:

$$\begin{aligned} & max\_rprec(\{x_1, \ldots, x_n\}) \\ = & \, max\{rprec(x_1), \ldots, rprec(x_n)\} \\ < & \, max\_rprec(\{y_1, \ldots, y_n\}) \end{aligned}$$

Let $rprec(a)$ be the precision of the integer variable a without taking signs into account. For example, a 32-bit signed integer a has $rprecision(a) = 31$ (the number of bits adding to its absolute maximum value).
If `operator?` is `operator<` or `operator<=`, the loop counter variable is likely to loop around when the afterthought is an increment. The same idea, with a flipped `operator?` and
$max\_prec(x1, \ldots, xn) > max\_rprec(y1, \ldots, yn)$
needs to cause a second-stage flag.

4. Checking reference indices.
   For each block, we have the lines that include references, as well as the variables used as indices. In the previous step, variables were second-stage flagged for being dangerous to use in indices, taking execution paths into account. The last step: check whether second-stage flags and index variables intersect. If this is the case, report a potential range mismatch.

3.5 Implementation details

3.5.1 File summary

The source code of Sans_t is available at
`https://github.com/Sanstee/Sans_t`.
One should use GCC version `9.2.0` to build the plugin.
It is required to run `./contrib/download_prerequisites` from the the top-level GCC source directory before building GCC, in order to build Sans_t.
The individual goals each of the files achieve can be summarized as follows:

src/genericparser.cc Parse the GENERIC representation of functions into a `BlockStorage` instance.

src/blockstorage.cc Represent a directed, cyclic graph with the `BlockStorage` class. Nodes represent basic blocks. Also keeps track of comparisons and local effects within blocks.

src/blockinterpreter.cc Propagate flags through a `BlockStorage` model to check for the type mismatch bug. Triggers a warning enabled by `-Wall` if the bug is detected.

src/sans_t_scoped.cc Registers the `plugin_finish_parse_function` callback.

For more precise implementation details, it is advised to inspect source code. The source code contains comments for almost every function. Comments describe parameters, return values, function summaries and implementation notes.

3.6 Sans_t example output

Using the libraries that are employed in the static analysis section of this study, a set of example outputs have been generated. The full output along with explanations can be found in section 4.1.8.

3.6.1 TweetNaCl

The plugin is created to find the bug present in the current version of TweetNaCl. It does exactly that when included whilst compiling TweetNaCl, resulting in a total of seven reports.

```
./tweetnacl-usable/tweetnacl.c: In function '
    crypto_sign_ed25519_tweet':
./tweetnacl-usable/tweetnacl.c:723:26: warning: Loop ctr var
    suspected range mismatch. [-Wall]
723 |    FOR(i,n) sm[64 + i] = m[i];
|                          ~^~~
```

The above is a report for a non-alarming instance of the bug with a precision difference of a single bit. The full report contains two warnings, one for each index containing the loop counter variable.

```
./tweetnacl-usable/tweetnacl.c: In function '
    crypto_sign_ed25519_tweet_open':
./tweetnacl-usable/tweetnacl.c:790:21: warning: Loop ctr var
    suspected range mismatch. [-Wall]
790 |    FOR(i,n) m[i] = sm[i];
|                     ~~^~~
```

This is one of the five reports for the `crypto_sign_ed25519_tweet_open` function, as depicted in Figure 3.6.
Thus, the bug is correctly warned about in TweetNaCl, without any false positives.

3.6.2 BearSSL, mbed TLS and Libsodium

The plugin output for the other three libraries consists solely of false positives. Each of these can be attributed to the fact that the plugin does not

perform value analysis; it solely traces the variables. The following commented report is representative of the type of false positives that can be expected whilst running the plugin in its current state:

```
ccm.c: In function 'ccm_auth_crypt':
ccm.c:146:36: warning: Loop ctr var suspected range mismatch. [-
    Wall]
146 |             (dst)[i] = (src)[i] ^ b[i];
ccm.c:287:9: note: in expansion of macro 'CTR_CRYPT'
287 |         CTR_CRYPT( dst, src, use_len );
```

The for-loop surrounding line 146: `for( i = 0; i < (len); i++ )`, inside the macro. The types are `unsigned char i` and `size_t tag_len`. The macro argument `use_len` is kept smaller or equal to `16` in line 278, by the following statement: `size_t use_len = len_left > 16 ? 16 : len_left`. Thus, the wider integer is guaranteed to have a value in the range of the byte.

Using the plugin on Libsodium does not generate a single report. Compiling BearSSL and mbed TLS with the plugin generates a combined total of ten false positives. Without counting warnings about the same line (e.g. array copies), this is reduced to six warnings.

3.7 Possible improvements

The current Sans_t prototype is functional to a certain extend. Whilst the output for TweetNaCl is as desired, unnecessary false positives remain present when Sans_t is used on the other cryptography libraries. For instance, problems arise when masking is used to put an integer in a safe range for comparison. Masking is a common technique within cryptography libraries. Therefore, adding a custom interpretation for expressions that mask integers can be considered feasible.
On the other hand, the flagging used in Sans_t can be made more generic than it currently is. Constructions leading to a first/second stage flag are hard-coded, due to the plugin focusing only on the type mismatch bug in TweetNaCl. The flagging system could be extended to flag according to a user-defined sequence of expressions, also requiring a variable amount of flagging stages. This would allow for more types of bugs to be searched for.
Some C syntax is not interpreted by the current prototype. For example, assignments inside the condition of conditional statements are ignored.
Lastly, one could consider using GIMPLE, a simplified subset of GENERIC, instead of GENERIC.
Sans_t shows it is possible to reliably detect the type mismatch bug in TweetNaCl. This currently comes at a cost of generating false positives, some of which can easily be made exceptions for by rewriting parts of the GENERIC parser.

Chapter 4

Static analysis of C cryptography libraries

In this chapter, analysis results of three static analyzers used on four C cryptography libraries will be discussed. The static analysis methodology that leads to the results is described. The true positives are examined closely. The false positives are used to construct a list of coding practices that lead to them being reported.

4.1 Static Analysis Methodology

In this section, the methodology and version details that may be used to reproduce the static analysis results are summarized. The results can be found from Section 4.2 onward.

4.1.1 Library versions

Library	Version	Release date
BearSSL	v0.6	2019-09-04
mbed TLS	2.16.3	2019-09-06
Libsodium	1.0.18	2019-05-31
TweetNaCl	20140427	2014-04-27

4.1.2 Static analyzer versions

Static analyzer	Version	Release date
Facebook Infer	v0.17.0	2019-08-06
Clang Static Analyzer*	6.0.0	2018-03-08
Splint	3.1.2	2018-02-20

*`scan-build` is a Perl script that invokes the Clang Static Analyzer.

4.1.3 OS and compiler versions

Every static analysis tool, except for the Sans_t plugin, was used under the following circumstances:

Operating system

```
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.3 LTS
Release:        18.04
```

GCC (default compiler)

```
gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
```

The target triple:

```
x86_64-linux-gnu
```

Clang

```
clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)
Target: x86_64-pc-linux-gnu
Thread model: posix
```

The Sans_t plugin is made for a newer version of GCC than the one used to compile the libraries whilst performing static analysis.
This newer version is referred to as `/gcc-install/bin/gcc` in any command; it uses `gcc (GCC) 9.2.0`. Tests ran using this version of GCC are performed on a secondary operating system.

Secondary OS

```
Distributor ID: Ubuntu
Description:    Ubuntu 19.10
Release:        19.10
```

The target triple:

```
x86_64-pc-linux-gnu
```

4.1.4 Infer

BearSSL `infer run -o ./BearSSLInferOut --no-uninit -- \`
`make`

mbed TLS `infer run -o ./mbedTLSInferOut --no-uninit -- \`
`make no_test`

Libsodium `./configure; infer run -o ./LibsodiumInferOut \`
`--skip-analysis-in-path test --no-uninit -- \`
`make`

TweetNaCL `infer run -o ./TweetNaClInferOut --no-uninit -- \`
`gcc tweetnacl.c tweetnacl.h randombytes.c randombytes.h main.c`

Note the usage of the `--no-uninit` flag. The uninitialized check generates a lot of false positives when enabled. For instance, it will count 34 instances of uninitialized values in TweetNaCl, which are mostly false positives caused by misinterpreting the `FOR` macro.

4.1.5 Clang Static Analyzer / scan-build

BearSSL
See Figure 4.1 on the next page.

mbed TLS
For mbed TLS, the same command as in Figure 4.1 is used, but with make target `no_test` instead of the default one.

Libsodium

1. `scan-build ./configure`
2. The same command as in Figure 4.1.

Running the `configure` script through `scan-build` is recommended by the scan-build recommended usage guidelines [31].

TweetNaCL
The same command as in Figure 4.1 is used, apart from using
`gcc tweetnacl.c tweetnacl.h randombytes.c randombytes.h main.c`
instead of `make`.

```
scan-build \
-enable-checker alpha.core.CallAndMessageUnInitRefArg \
-enable-checker alpha.core.CastToStruct \
-enable-checker alpha.core.PointerArithm \
-enable-checker alpha.core.SizeofPtr \
-enable-checker alpha.deadcode.UnreachableCode \
-enable-checker alpha.security.ArrayBoundV2 \
-enable-checker alpha.security.MallocOverflow \
--keep-empty --force-analyze-debug-code \
make 2>clangstderr.txt
```

Figure 4.1: scan-build command used for BearSSL.

4.1.6 Splint

BearSSL `splint src/aead/ccm.c -Iinc -Isrc &>bear_ccm.txt`

mbed TLS

`splint library/ccm.c -Iinclude -preproc &>mbed_ccm.txt`

Libsodium `splint \`
`src/libsodium/crypto_aead/xchacha20poly1305/sodium/\`
`aead_xchacha20poly1305.c -Isrc/libsodium/include/sodium \`
`&>libsodium_aead_xchacha20poly1305.txt`

TweetNaCl `splint tweetnacl.c -I. &>tweetnacl_tweetnacl.txt`

Because of the high amount of false positives Splint produces, Splint is only used on a the following unannotated files, along with any files included by them:

BearSSL `src/aead/ccm.c`

mbed TLS `library/ccm.c`

Libsodium `src/libsodium/crypto_aead/xchacha20poly1305`
`/sodium/aead_xchacha20poly1305.c`

TweetNaCl `tweetnacl.c`

4.1.7 CBMC

CBMC is one of the original tools that looked promising for the static analysis of cryptography libraries. Whilst this may still be true, errors have presented itself whilst trying to compile libraries with the `goto-cc` compiler.

A valid GOTO-binary is essential for verification with CBMC. For example, building Libsodium according to the instructions found in the CPROVER manual [7]:

```
./configure YACC=byacc CC=goto-cc --host=none-none-none
make
```

Running these commands errors during the make process. The output:

```
[1msodium/utils.c:[0m In function [1m'sodium_memzero'[0m:
[1msodium/utils.c:112:1: [31merror:
[0msyntax error before `len'
if (len > 0U && memset_s(pnt, (rsize_t) len,
0, (rsize_t) len) != 0) {
PARSING ERROR
make[3]: *** [sodium/libsodium_la-utils.lo] Error 1
make[2]: *** [all-recursive] Error 1
make[1]: *** [all-recursive] Error 1
make: *** [all-recursive] Error 1
```

Apart from TweetNaCl, none of the libraries compiled with the `goto-cc` compiler. Therefore, CBMC was skipped altogether.

4.1.8 Sans_t

BearSSL

```
make \
CFLAGS='-fplugin=/ABSOLUTE/PATH/sans_t_scoped.so -Wall -fPIC' \
CC='/gcc-install/bin/gcc'
```

mbed TLS

```
make no_test \
CFLAGS='-fplugin=/ABSOLUTE/PATH/sans_t_scoped.so -Wall' \
CC='/gcc-install/bin/gcc'
```

Libsodium

```
./configure; make \
CFLAGS='-fplugin=/ABSOLUTE/PATH/sans_t_scoped.so -Wall' \
CC='/gcc-install/bin/gcc'
```

TweetNaCl

```
/gcc-install/bin/gcc \
tweetnacl.c tweetnacl.h randombytes.c randombytes.h main.c \
-fplugin=../sans_t_scoped.so -Wall
```

4.2 True Positives

```
src/ec/ec_p256_m62.c:582: error: DEAD_STORE
The value written to &cc (type unsigned long) is never used.
580.     w = t[2] - cc;
581.     t[2] = w & MASK52;
582. >   cc = w >> 63;
583.     w = t[3] - BIT(36);
584.     t[3] = w & MASK52;
```

Listing 4.1: A report about a dead store in BearSSL, generated by Infer. The Clang Static Analyzer also reports this bug.

The code is located in the `f256_final_reduce` function, which performs a conditional subtraction on its input value. The function ensures that the output value is smaller than the constant $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. `uint64_t cc` acts as the carry for the subtraction of `2^192 + 2^96` from `t`. The subtraction uses four limbs of out five. `t[2]` only needs to subtract the carry from the subtraction of `2^96`. This should be able to produce a carry, which is calculated, but not subtracted in `w = t[3] - BIT(36)`. The carry `cc` is either `0` or `1`, with the shift of `uint64_t w` being a logical shift.

The bug was reported, and quickly fixed afterwards. The carry is now correctly subtracted from the fourth limb, as shown in Figure 4.2.
The cause of the bug is likely human error, as the carry is propagated correctly for the other limbs. The bug seems like a mistake that should generate a compiler warning, but it does not in GCC. The following command compiles the file without any warnings whatsoever:

```
gcc -Wall -Wextra -Isrc -Iinc \
-c -o build/obj/ec_p256_m62.o src/ec/ec_p256_m62.c
```

Clang does not warn about the unused carry either, with `-Weverything` enabled.
Another issue that prevents the bug from being detected more easily is the fact that it only causes a wrong result for very specific inputs. Randomized tests have an estimated chance of 2^{-78} to both trigger the bug and to carry on using the incorrect result.

```
-     w = t[3] - BIT(36);
+     w = t[3] - BIT(36) - cc;
```

Figure 4.2: The carry propagation bugfix [27]

```
library/bignum.c:1387: error: DEAD_STORE
The value written to &t (type unsigned long) is never used.
1385.   #endif /* MULADDC_HUIT */
1386.
1387. >     t++;
1388.
1389.       do {
```

Listing 4.2: A report about a dead store in mbed TLS, generated by Infer.

The code is located in the `mpi_mul_hlp` function, a helper function for `mbedtls_mpi` multiplication.
`mbedtls_mpi_uint c = 0, t = 0` is used to initialize `c` and `t`. Both are used in the assembler instructions from `bn_mul.h`; separate routines are used depending on whether `MULADDC_HUIT` (in `bn_mul.h`) is defined. The type definition for `mbedtls_mpi_uint` is either
`typedef uint64_t mbedtls_mpi_uint` or
`typedef uint32_t mbedtls_mpi_uint`, as specified in `bignum.h`.
Using a post-increment on a local, non-static variable seems unnecessary. The bug is not harmful and the line is likely a code remnant.

```
ecdsa.c:300:13: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
if( *p_sign_tries++ > 10 )
^~~~~~~~~~~~~~~
ecdsa.c:300:14: warning: Pointer arithmetic on non-array
    variables relies on memory layout, which is dangerous
if( *p_sign_tries++ > 10 )
^~~~~~~~~~~~~~
ecdsa.c:313:17: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
if( *p_key_tries++ > 10 )
^~~~~~~~~~~~~~
ecdsa.c:313:18: warning: Pointer arithmetic on non-array
    variables relies on memory layout, which is dangerous
if( *p_key_tries++ > 10 )
^~~~~~~~~~~~~
4 warnings generated.
```

Listing 4.3: A report about out of bound memory access in mbed TLS, generated by the Clang Static Analyzer.

The code is located in the `ecdsa_sign_restartable` function. The pointer `p_sign_tries = &rs_ctx->sig->sign_tries` is within bounds in the first do-while iteration. However, the order of the left operand of the `>` operator seems to be illogical. `*p_sign_tries++` increments the address that `p_sign_tries` points to for the next iteration of the do-while loop; it then dereferences the pointer. The author likely meant to do `(*p_sign_tries)++`, in which case it acts as a counter. `(*p_sign_tries)++` would increment the value `*p_sign_tries` every do-while iteration.

The bug seems like another human error, repeated multiple times for similar code. The conditions that trigger a second do-while loop, which causes undefined behavior, are enumerated below:

1. `r = xR mod n = 0`.
2. `s = (e + r * d) / k = t (e + rd) / (kt) mod n = 0`.

The chance of one of these conditions evaluating to true is incredibly small. Even if these were to evaluate to true, accessing the exceeding memory may trigger one of two events:

1. If the exceeding memory causes `*p_key_tries++ > 10` to evaluate to true, an early exit/cleanup is triggered.
2. Otherwise another attempt to generate a valid ephemeral secret is made.

If the whole signature needs to be repeated, a similar scenario occurs.
Just like the carry-propagation omission, the odds of triggering the bug in ordinary usage are incredibly low. Randomized tests are unlikely to catch it, but specifically testing for edge cases could help.

4.3 Common reports

In this section, the frequencies of the several types of reports are shown for Infer and the Clang Static Analyzer. Additionally, coding practices that lead to the bulk of these reports are listed. Code examples are provided for most of the report types.

4.3.1 Infer

The frequency of all of the report types that where found by Infer can be found in Table 4.1 on the next page. Note that the `--no-uninit` flag was used. Figures that show simplified examples of code segments that trigger the reports, are shown on the next pages. The following are general coding practices leading to a cluttered Infer report:

Dead store

1. Moving a pointer one final time after reaching the end of a buffer. An example is shown in Figure 4.3.
2. Assigning values to return variables that are guaranteed to be overwritten. An example is shown in Figure 4.4.
3. Storing but not using return values. An example is shown in Figure 4.5.
4. Zeroing storage after usage. An example is shown in Figure 4.6.

Memory leak

1. Data with uncommon lookup schemes (e.g. named data). An example is shown in Figure 4.7. In this example, `const char *oid` is used to look up specific memory addresses.

Null Dereference

1. Possibly calling functions like `memcpy` with arguments that are (potentially both) `null`, whilst the `memcpy` size argument is `0`. This is undefined behavior according to section 7.24.1.2 of the C11 draft [18]. An example is shown in Figure 4.8.
2. Not checking for `null` arguments, e.g. in functions that are only called a few times with safe arguments. An example is shown in Figure 4.9.

Report	BearSSL	mbed TLS	Libsodium	TweetNaCl
Dead store	6	42	5	0
Memory leak	0	8	1	0
Null dereference	8	2	0	0
Total	**14**	**52**	**6**	**0**

Table 4.1: Frequency of the reports from Infer.

```
1  // Assuming s is initialized and
2  // both contain at least 2 integers.
3  void cpytwo(int *d, int *s){
4      *d++ = *s++;
5      *d++ = *s++;
6  }
```

Figure 4.3: The value written to `&s` (type `int*`) in line 5 is never used.

```
1  int func(int x){
2      int ret = EXIT_FAILURE;
3      switch (x)
4      {
5          case 1:
6              ret = otherfunc();
7              break;
8          case 2:
9              ret = EXIT_SUCCESS;
10             break;
11         default:
12             ret = EXIT_FAILURE;
13     }
14     return ret;
15 }
```

Figure 4.4: The value written to `&ret` (type `int`) in line 2 is never used.

```
1  void func(int x){
2      int i = function_with_side_effects(&x);
3      for(i = 0; i < x; ++i)
4          ;
5      // etc
6  }
```

Figure 4.5: The value written to `&i` (type `int`) in line 2 is never used.

```
1  static inline int
2  crypto_verify_n(const unsigned char *x_,
3                  const unsigned char *y_,
4                  const int n)
5  {
6      const    __m128i zero = _mm_setzero_si128();
7      volatile __m128i v1, v2, z;
8      /*
9      * Verification using v1, v2 and z
10     * (implementation left out)
11     */
12     v1 = zero; v2 = zero; z = zero;
13     return (int) (((uint32_t) m + 1U) >> 16) - 1;
14 }
```

Figure 4.6: A simplified example derived from the `crypto_verify_n` function in Libsodium, located in the file `verify.c`. The value written to `&v1` (type `void`) in line 12 is never used.

```
/**
* \brief  Create or find a specific named_data entry for
*         writing in a sequence or list based on the OID.
*         If not already in there, a new entry is added
*         to the head of the list.
* \return A pointer to the new / existing entry on success.
* \return NULL if if there was a memory allocation error.
*/
mbedtls_asn1_named_data *mbedtls_asn1_store_named_data(
    mbedtls_asn1_named_data **list,
    const char *oid, size_t oid_len,
    const unsigned char *val,
    size_t val_len
);
```

Figure 4.7: A simplified function declaration derived from the `mbedtls_asn1_store_named_data` function in mbed TLS, located in the file `asn1_write.c`.

```
#include <string.h>
int main(int argc, char *argv[]){
    memcpy(NULL, NULL, 0);
}
```

Figure 4.8: Null argument where non-null required (line 3).

```
1  typedef enum {A, B} AB;
2  extern const int a;
3  extern const int b;
4
5  static const int *
6  id_to_int_ref(AB c)
7  {
8      switch (c) {
9          case A:
10             return &a;
11         case B:
12             return &b;
13     }
14     return NULL;
15 }
16
17 int func(AB c)
18 {
19     const int *d;
20
21     d = id_to_int_ref(c);
22     return *d;
23 }
```

Figure 4.9: A function similar to the `api_generator` function in BearSSL, located in the file `ec_prime_i15.c`. Pointer `d` last assigned on line 21 could be null and is dereferenced at line 22.

4.3.2 Clang Static Analyzer

Table 4.2 on the next page shows the frequency of reports from the Clang Static Analyzer. Figures that show simplified examples of code segments that trigger the reports, are shown on the next pages. The following are general coding practices leading to a cluttered report by the Clang Static Analyzer:

Unreachable code

1. Checking the return value of functions that handle errors by themselves (through an early exit). An example is shown in Figure 4.10.
2. Code guarded by a conditional statement that depends on a volatile integer. Such cases are not easily reproduced, and the simple ones such as the case depicted in Figure 4.11 are interpreted correctly. However, cases such as the ones shown on pages 71 and 76 in the appendix do show up.

Dead Assignment

1. Writing values to a return status variable that is guaranteed to be overwritten. An example is shown in Figure 4.4.
2. Storing the result of a function without using it. An example is shown in Figure 4.5.

Cast from non-struct type to struct type

Casting a <pointer to a <pointer to a structure> to a <pointer to another structure>. This is common practice in object oriented C code and safe, but the Clang Static Analyzer acts as if both types are struct. An example is shown in Figure 4.12.

Out-of-bounds access

1. This report was triggered a lot by automatically generated code, which manages resources differently than code written by humans. Cases in the appendix that contain examples of this report can be found on pages 57 and 61.
2. Constructions in which the size of an array is not immediately clear to the analyzer. For example, saving leftover bytes, an amount lower than the block size, until a new block can be filled. The analyzer does not recognize there is an implicit size constraint on the leftover byte-array parameter. Examples of this report can be found on the pages 74 and 64 in the appendix.

Uninitialized argument value

The analyzer does not detect implicit bounds to arguments that are either presented in documentation or in comments. This causes the analyzer to skip loop iterations, among other conditional constructions. If these contain variable initialization(s), they are marked as potentially uninitialized. Using these variables as function arguments causes this report. An example of this report can be found in the appendix on page 57.

Group	Report	BearSSL	mbed TLS	Libsodium	TweetNaCl
API	Argument with 'nonnull' attribute passed null	1	1	0	0
Dead code	Unreachable code	6	14	8	0
Dead store	Dead assignment	3	12	0	0
	Dead increment	0	1	0	0
Logic error	Cast from non-struct type to struct type	30	0	0	0
	Dangerous pointer arithmetic	5	2	2	0
	Division by zero	2	0	0	0
	Out-of-bound access	18	21	6	0
	Potential unintended use of sizeof() on pointer type	0	0	1	0
	Result of operation is garbage or undefined	0	3	0	0
	Uninitialized argument value	6	14	0	0
Unix API	malloc() size overflow	0	0	1	0
Memory error	Bad free	0	1	0	0
Total		**71**	**69**	**18**	**0**

Table 4.2: Frequency of the reports from the Clang Static Analyzer.

```c
#include <stdlib.h>

void early_exit(){
    // possibly cleanup code
    exit(1);
}

int func2(int a, int b)
{
    if(a) early_exit();
    // code that can run safely now
    if(b) early_exit();
    // etc...
    return 0;
}

int func(int a, int b)
{
    if(func2(a, b) != 0)
        early_exit();
    // normal execution
    return 0;
}
```

Figure 4.10: The statement on line 20 is never executed.

```c
static volatile int a = 0;

int main(){
    if(a == 0) {
        exit(1);
    }
    return 0;
}
```

Figure 4.11: No bugs found.

```
1  typedef struct class_ class;
2  struct class_{
3      int x;
4  };
5
6  typedef struct{
7      class *c;
8  } class_ext;
9
10 void get_ext(class **ctx){
11     class_ext *xc = (class_ext *)ctx;
12 }
```

Figure 4.12: Casting a non-structure type to a structure type and accessing a field can lead to memory access errors or data corruption (line 11). The example is derived from the `kk_get_pkey` function in BearSSL, located in the file `x509_knownkey.c`.

4.3.3 Splint

Splint report frequencies are intentionally excluded from this section. This is due to the high amount of reports generated by Splint on unannotated C code. The amount would not do justice to Splint when it is used continuously throughout the development process with annotations.
In a study comparing static analyzers including Splint, similar results were obtained. The study includes tests that were ran with several flags, filtering out uninteresting output. A statement regarding the performance of Splint is made in this study by Moerman et al. [25]:

"Splint's analysis resulted in output that is time-consuming to analyze and it failed to parse several system library headers rendering it useless in many real-world scenarios."

The following are general coding practices leading to a cluttered Splint report. Figures that show simplified examples of code segments that trigger the reports, are shown on the next page.

1. Not specifying every value inside an initialization block. An example is shown in Figure 4.13.

2. Not checking whether the right operand of a bit shift might be negative. An example is shown in Figure 4.14.

3. Using masks to assure variables are in a safe range before usage. An example is shown in Figure 4.15.

These are the reports that should still occur if annotations were used (but flags can be used to inhibit warnings).

```
int main(){
    int a[10] = {0};
    return 0;
}
```

Figure 4.13: Initializer block for a has 1 element, but declared as `int [10]`

```
// assuming this function is called very rarely
// with values that are safe.
int func(unsigned int a, int c){
    return a << c;
}
```

Figure 4.14: The right operand to a shift operator may be negative (behavior undefined).

```
#include <limits.h>

int main(){
    signed int a = -10;
    unsigned char b = 1 << (a & 7);
    return 0;
}
```

Figure 4.15: The right operand to a shift operator may be negative (behavior undefined).

4.4 Cryptography code challenges

The previous section shows coding practices that lead to the bulk of the false positives that were encountered. The reports contain a few false positives that one would encounter less in regular C code. These contain – but are not limited to – the following practices:

Zeroing storage

Zeroing storage previously used for (semi) sensitive values is especially important in cryptography libraries. Surprisingly, reports on this issue were relatively low, as zeroing storage after usage could be reported as a dead assignment.

Argument safety assumption

Some functions are not made to be reused, and therefore assume a certain safety in terms of the arguments it might expect. More concretely, functions tend not to check each argument for being in the right/expected range, as they are only ever called by functions from which the argument value ranges are known. To a static analyzer such as the Clang Static Analyzer, these functions are especially prone to false positives. An example is the `window_to_affine` function in BearSSL. The Clang Static Analyzer assumes any `int num` argument may be supplied, while the author clearly commented the following:
"`This function works for windows up to 32 elements`".
Not only does this generate a lot of false positives, but the types of false positives produced by wrongfully assuming certain branches can be skipped varies greatly. This results in output that is time-consuming to process.

Volatile integers

Volatile integers are definitively not specific to cryptography code, but they do play a role in some libraries. In the Clang Static Analyzer reports, the qualifier seems to cause branches to be wrongfully deemed unreachable at times. Similarly, certain SIMD vector instructions cause reports about dangerous pointer arithmetic.

Static analysis of regular C code seems generally as effective as the static analysis of cryptography code. Both Infer and the Clang Static Analyzer have found true positives that relate to simple human error, rather than an obscure edge case that may be specific to cryptography code.
The extra reports specific to cryptography code do not mean cryptography code necessarily generates more false positives than any other type of C code. More important are the true positives, which can possibly have far greater consequences than a small error in a regular C program. For the amount of true positives generated, one could reasonably say these are worth going over the high amount of false positives.

Chapter 5

Related Work

Static analysis is a field concerned with analyzing code without running it. There are a lot techniques to do this, as well as problems to solve using static analysis. Formal methods includes data-flow analysis [20] and Hoare logic [16]. Separation logic [30] is an extension to Hoare logic that supports reasoning about pointer data structures and transfer of ownership in concurrent programs.

Static-analysis tools using these techniques can be applied to cryptography in a number of ways. For one, there is a work that proves the functional equivalence of two AES implementations using CBMC [28]. The work focuses on cryptographic primitives and their equivalence, unlike this thesis, which analyzes complete libraries and does not try to prove the functional equivalence of implementations.

There are works comparing the static analysis tools that are used in this study, such as the Clang Static Analyzer and Facebook Infer, against each other [3] [25]. These studies do not limit the code that is analyzed to code found in cryptography libraries, like in this thesis. The studies use test suites, whilst this thesis uses real-world code.

A work that attempts to address the misuse of cryptographic primitives through static analysis exists [29]. The study introduces a static taint analysis engine called TaintCrypt. TaintCrypt aims to detect dangerous flows. The study is not concerned with detecting a specific bug like the type mismatch bug that is analyzed in this thesis.

It is worth mentioning that around the time of writing this, NIST held the sixth Static Analysis Tool Exposition (SATE). Due to the report not being published yet, the edition V is the most recent report accessible [10]. The SATE reports use test suites for static analyzer comparison, whereas this thesis uses real-world code.

Articles that perform static analysis on cryptography libraries in a non-academic setting are also noteworthy [21] [5]. These works mainly use source code with other purposes, or popular cryptography libraries like OpenSSL.

This thesis uses some less popular cryptography libraries, does not focus on only a single library/static analyzer, and contains (likely) explanations for most of the reports (found in the appendix).

The Sans_t plugin uses techniques categorized as data-flow analysis. However, at the time of implementing these techniques in Sans_t, this was not apparent. Most techniques naturally followed from what was theorized in the strategy and basic notions from control-flow analysis. The fundamentals of control flow analysis, such as basic blocks and directed graph, can be found in [1]. An approach for transforming abstract syntax trees to control flow graphs is presented in [32], albeit more focused on logical operators in C than this study. This study uses a similar technique to achieve a model that is helpful for analysis. The usage of the GCC plugin system for static analysis can be traced back to tools such as Dehydra [14]. It should be noted this project has been abandoned in favour of a Clang-based analyzer.

Chapter 6

Conclusions

Facebook Infer and the Clang Static Analyzer have proven to be helpful tools for finding bugs in C cryptography libraries. Specialized functions that are only called with implicitly safe arguments are the root cause for extra false positives using the Clang Static Analyzer. Such constructions are commonly used in cryptography libraries. The three true positives detected in this study are clear human error, and one would expect these to be caught by a compiler. The amount of false positive for these two tools is acceptable. Using them is recommendable for major-release candidates.
Sans_t is able to detect the bug in TweetNaCl. The GCC plugin reports a couple of false positives for other cryptography libraries. Special interactions for masking and ternary operators could fix these. Lastly, certain C constructions are not parsed correctly by Sans_t, possibly lowering the amount of reports. The source code of Sans_t is available at `https://github.com/Sanstee/Sans_t`.

Bibliography

[1] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[2] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 2015.

[3] Andrei Arusoaie, Stefan Ciobâca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 161–168. IEEE, 2017.

[4] Daniel J. Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl: A crypto library in 100 tweets. In D.F. Aranha and A. Menezes, editors, *Progress in Cryptology - LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 64–83. Springer, 2015. ISBN 978-3-319-16294-2.

[5] Sam Blackshear and Dino Distefano. Finding inter-procedural bugs at scale with Infer static analyzer, Jun 2018. `https://engineering.fb.com/android/finding-inter-procedural-bugs-at-scale-with-infer-static-analyzer/` (accessed 2020-01-01).

[6] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[7] `https://www.cprover.org/cprover-manual/goto-cc/` (accessed 2020-01-01).

[8] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160, December 3 2014. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160` (accessed 2020-01-01).

[9] Joeri De Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, 2015.

[10] Aurelien M. Delaitre, Bertrand C. Stivalet, Paul E. Black, Vadim Okun, Terry S. Cohen, and Athos Ribeiro. SATE V Report: Ten Years of Static Analysis Tool Expositions. Technical report, NIST, 2018. `https://www.nist.gov/publications/sate-v-report-ten-years-static-analysis-tool-expositions` (accessed 2020-01-01).

[11] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. New Mexico State University Portland, OR, 2003.

[12] David Evans and David Larochelle. *Splint Manual*. Secure Programming GroupUniversity of Virginia Department of Computer Science. `https://splint.org/manual/manual.pdf` (accessed 2019-12-19).

[13] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51, 2002.

[14] Taras Glek. DeHydra Source Analysis Tool, 2007. `https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Dehydra` (abandoned; accessed 2020-01-01).

[15] GNU Compiler Collection (GCC). `https://github.com/gcc-mirror/gcc` (accessed 2019-10-02), 2019.

[16] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[17] ISO. *ISO/IEC 9899:1990: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1990.

[18] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[19] ISO. *ISO/IEC 9899:2017: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 2017.

[20] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[21] Dave Kleidermacher. Using static analysis to detect coding errors in open source security-critical server applications, Mar 2014. `https://www.embedded.com/using-static-analysis-to-detect-coding-errors-in-open-source-security-critical-server-applications/` (accessed 2020-01-01).

[22] Ted Kremenek. Finding software bugs with the clang static analyzer. *Apple Inc*, 2008.

[23] Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In E Ábrahám and K Havelund, editors, *TACAS 2014 (ETAPS)*, volume 8413 of *LNCS*, pages 389–391. Springer, Heidelberg, 2014.

[24] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. *ACM SIGPLAN Notices*, 51(6):1–15, 2016.

[25] Jonathan Moerman. Evaluating the performance of open source static analysis tools. Bachelor's thesis, Radboud University Nijmegen, Nijmegen, The Netherlands, 2018.

[26] NaCl: Networking and Cryptography library: Validation and verification. version 2016.06.17 of the valid.html web page. `https://nacl.cr.yp.to/valid.html` (accessed 2019-09-18).

[27] Thomas Pornin. Fixed carry propagation bug in P-256 'm62' implementation, Dec 2019. `https://bearssl.org/gitweb/?p=BearSSL;a=commitdiff;h=252dba914912e694d0e69754f0167060fc4d2ba6` (accessed 2020-01-01).

[28] Hendrik Post and Carsten Sinz. Proving functional equivalence of two AES implementations using bounded model checking. In *2009 International Conference on Software Testing Verification and Validation*, pages 31–40. IEEE, 2009.

[29] Sazzadur Rahaman and Danfeng Yao. Program analysis of cryptographic implementations for security. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 61–68. IEEE, 2017.

[30] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[31] scan-build: running the analyzer from the command line. `https://clang-analyzer.llvm.org/scan-build.html` (accessed 2019-09-11).

[32] Naftali Schwartz. Steering clear of triples: Deriving the control flow graph directly from the Abstract Syntax Tree in C programs. Technical report, New York University New York United States, 1998.

[33] Emma Söderberg, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming*, 78(10):1809–1827, 2013.

[34] Splint manual - memory management. `https://splint.org/manual/html/htmlBak3/sec5.html` (accessed 2019-12-19).

[35] Richard M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals*, 2019. Manual document for the GCC 10.0.0 (pre-release). `https://gcc.gnu.org/onlinedocs/gccint.pdf` (accessed 2020-01-01).

[36] Abhijat Vichare. The Conceptual Structure of GCC. *Indian Institute of Technology, Bombay*, 2008. `https://www.cse.iitb.ac.in/grc/intdocs/gcc-conceptual-structure.html#toc_Copyright` (accessed 2019-10-26).

[37] Benjy Weinberger, Craig Silverstein, Gregory Eitzmann, Mark Mentovai, and Tashana Landray. Google C++ style guide. *Section: Function Parameter Ordering*, 2013. Updated version found at `https://google.github.io/styleguide/cppguide.html#Output_Parameters` (accessed 2020-01-01).

[38] David A. Wheeler. How to prevent the next Heartbleed, Jan 2017. `https://dwheeler.com/essays/heartbleed.html` (accessed 2020-01-01).

Appendix A

Appendix

A.1 Static Analyzer Output

A.1.0 Report format

The format that is used for the reports is:

```
Tool output for which no completely similar case has
    been presented.
```

[FP/FP] With a comment underneath summarizing why something is reported and/or what happens. To personally verify any of the reports, it is advised to check the source code, *as only small snippets will be added in the comments, which are meant to provide guidance through the source code.* A couple of additional notes on the reports:

- Some reports may seem out of place, for instance the ones on code that is not in the library. These are included for completeness.
- TP stands for True Positive and FP stands for False Positive. These terms are not completely applicable to some reports, especially to the reports that hint something *could* be wrong (e.g. using `sizeof()` on a pointer). In these cases, such a statement will be treated as a claim in order to label it, but it should be noted that the tool is not necessarily wrong in these scenarios.

A.1.1 Infer

BearSSL

```
tools/files.c:62: error: NULL_DEREFERENCE
  pointer `vbuf.buf` last assigned on line 37 could be null and
     is dereferenced by call to `xblobdup()` at line 62, column
     10.
  60.                                       return NULL;
  61.                                   }
  62. >                                 buf = VEC_TOARRAY(vbuf);
  63.                                   *len = VEC_LEN(vbuf);
  64.                                   VEC_CLEAR(vbuf);
```

FP: Line 37: The initialization `bvector vbuf = VEC_INIT` correctly initializes the vector of bytes using the macro `#define VEC_INIT { 0, 0, 0 }`. The first member of a `bvector` struct is the pointer `type *buf`. The value could be `null` when the function `xblobdup` is called. The function `xblobdup` makes the following calls:

```
buf = xmalloc(len);
memcpy(buf, src, len);
```

Note that the `xmalloc` implementation contains the following check:

```
if (len == 0) {
    return NULL;
}
```

Assuming that the `len` argument, which is also stored in the `bvector` struct, is correct (meaning `0` when `vbuf.buf` is `null`), the `vbuf.buf` struct member should not be dereferenced. Technically this is undefined behavior.

```
src/codec/pemenc.c:170: error: DEAD_STORE
  The value written to &d (type char*) is never used.
  168.
  169.          /* Final zero, not counted in returned length.
     */
  170. >        *d ++ = 0x00;
  171.
  172.          return dlen;
```

FP: The pointer `unsigned char *d` is used to write bytes to `void *dest`. The final post-increment is for consistency and possibly for future additions to the code.

```
src/int/i62_modpow2.c:168: error: DEAD_STORE
  The value written to &lo (type unsigned long) is never used.
  166.                    f = MUL62_lo(d[0] + MUL62_lo(x[u], y[0])
      , m0i) << 2;
  167.
  168. >                  FMA2(hi, lo, xu, y[0], f, m[0], d[0] <<
     2, 0);
  169.                    r = hi;
  170.
```

FP: The low word `uint64_t lo` is indeed discarded in another `FMA2` call four lines later, whilst the high word is saved in `uint64_t r`. This the carry to be used for the multiplication of `y[v + 0]` for `size_t v = 1`. This way, the author does not need to write a separate function just because one of the results from the function `FMA2` is not needed in this single instance.

```
src/rsa/rsa_i15_keygen.c:566: error: DEAD_STORE
  The value written to &tlen (type unsigned long) is never used.
  564.                    q[plen] = 0;
  565.                    t ++;
  566. >                  tlen --;
  567.            }
  568.            br_i15_zero(t, p[0]);
```

FP: `uint16_t *t` is the pointer that was last assigned `t = q + 1 + qlen`. `q[plen] = 0` is written to `t`, requiring the variable `tlen` to reposition/decrease. Doing so is good practice and keeps the possibility of future additions to the code in mind.

```
src/rsa/rsa_i31_keygen_inner.c:591: error: DEAD_STORE
  The value written to &tlen (type unsigned long) is never used.
  589.                    q[plen] = 0;
  590.                    t ++;
  591. >                  tlen --;
  592.            }
  593.            br_i31_zero(t, p[0]);
```

FP: Same report for the "i31 engine". Henceforth, similar reports for different engines will be skipped.

```
tools/skey.c:483: error: DEAD_STORE
  The value written to &ret (type int) is never used.
  481.                    return 0;
  482.            }
  483. >          ret = 1;
  484.            switch (br_skey_decoder_key_type(&dc)) {
  485.                    const br_rsa_private_key *rk;
```

FP: The variable `int ret` is always overwritten in one of the switch cases. The variable `ret` is assigned `1` whenever the key type is recognized and the print functions are successful. If the key is not recognized or printing went wrong, `0` is returned. Initializing the automatic variable before usage is a

good choice, as these have an indeterminate initial value. One might argue `0` would have been a better choice, as `1` signals a successful exit.

```
src/ec/ec_p256_m62.c:582: error: DEAD_STORE
  The value written to &cc (type unsigned long) is never used.
  580.          w = t[2] - cc;
  581.          t[2] = w & MASK52;
  582. >        cc = w >> 63;
  583.          w = t[3] - BIT(36);
  584.          t[3] = w & MASK52;
```

TP: The variable `uint64_t cc` acts as the carry for the subtraction of `2^192 + 2^96` from `t`, which uses four limbs of out five. `t[2]` only needs to subtract the carry from the subtraction of `2^96`. This should be able to produce a carry, which is calculated, but not subtracted in `w = t[3] - BIT(36)`. The carry `cc` is either `0` or `1`, with the shift of `uint64_t w` being a logical shift.

```
src/ec/ec_prime_i15.c:705: error: NULL_DEREFERENCE
  pointer `cd` last assigned on line 704 could be null and is
     dereferenced at line 705, column 9.
  703.
  704.          cd = id_to_curve_def(curve);
  705. >        *len = cd->generator_len;
  706.          return cd->generator;
  707.    }
```

FP: This code is for the API, and the pointer `cd` will be null if one supplies an unrecognized `curve` to `id_to_curve_def`.

```
src/ec/ec_prime_i15.c:715: error: NULL_DEREFERENCE
  pointer `cd` last assigned on line 714 could be null and is
     dereferenced at line 715, column 9.
  713.
  714.          cd = id_to_curve_def(curve);
  715. >        *len = cd->order_len;
  716.          return cd->order;
  717.    }
```

FP: The same issue as in the previous report, but for requesting the order of the curve definition.

```
tools/names.c:981: error: NULL_DEREFERENCE
  pointer `suites.buf` last assigned on line 950 could be null
     and is dereferenced by call to `xblobdup()` at line 981,
     column 6.
  979.                  fprintf(stderr, "ERROR: no cipher suite
     provided\n");
  980.          }
  981. >        r = VEC_TOARRAY(suites);
  982.          *num = VEC_LEN(suites);
  983.          VEC_CLEAR(suites);
```

FP: Line 950: `VECTOR(cipher_suite) suites = VEC_INIT`.
A report similar to the first one. The vector initialization using the `VEC_INIT` macro initializes the struct member `suites.buf` as `null`, but the pointer is not dereferenced if the `len` struct member is also `0` when the function `xblobdup` is called.

mbed TLS

```
programs/x509/req_app.c:75: error: DEAD_STORE
  The value written to &ret (type int) is never used.
  73.   int main( int argc, char *argv[] )
  74.   {
  75. >     int ret = 1;
  76.       int exit_code = MBEDTLS_EXIT_FAILURE;
  77.       unsigned char buf[100000];
```

FP: The line `int ret = 1` is used to store the return values of two functions, to check these for errors. `ret` is an automatic variable, meaning explicit initialization is wise. Cases that branch into early exits are `ret != 0` and `ret == -1`. A potentially more suitable initial value would be `-1`, as this value triggers both early exits unless it is overwritten.

```
library/base64.c:123: error: DEAD_STORE
  The value written to &src (type unsigned char const *) is
     never used.
  121.       {
  122.           C1 = *src++;
  123. >         C2 = ( ( i + 1 ) < slen ) ? *src++ : 0;
  124.
  125.           *p++ = base64_enc_map[(C1 >> 2) & 0x3F];
```

FP: The unencoded `const unsigned char *src` buffer is being read from. The final post-increment is done for consistency.

```
library/x509_create.c:225: error: MEMORY_LEAK
  memory dynamically allocated by call to `
     mbedtls_asn1_store_named_data()` at line 218, column 17 is
     not reachable after line 225, column 5.
  223.
  224.       cur->val.p[0] = critical;
  225. >     memcpy( cur->val.p + 1, val, val_len );
  226.
  227.       return( 0 );
```

`mbedtls_asn1_named_data *cur` is a local pointer. Its last assignment is

```
cur = mbedtls_asn1_store_named_data( head, oid, oid_len,
                                     NULL, val_len + 1 )
```

As the name of the type suggest, this is named data. The named data is looked up with the `const char *oid` parameter, the object id. The `NULL` value causes the function

```
mbedtls_asn1_named_data *
mbedtls_asn1_store_named_data( mbedtls_asn1_named_data **head,
                               const char *oid, size_t oid_len,
                               const unsigned char *val,
                               size_t val_len)
```

not to write the pointer `val` to `cur->val.p`. Thus, the memory is still available to the caller of the function `mbedtls_x509_set_extension` because all of the arguments (including `oid`) are.

```
library/camellia.c:494: error: DEAD_STORE
  The value written to &RK (type unsigned int*) is never used.
  492.        *RK++ = *SK++;
  493.        *RK++ = *SK++;
  494. >      *RK++ = *SK++;
  495.
  496.    exit:
```

FP: Again, a post-increment to advance to the non-copied part of the buffer (reported two times as both pointers advance).

```
library/entropy.c:466: error: DEAD_STORE
  The value written to &ret (type int) is never used.
  464.    int mbedtls_entropy_write_seed_file(
      mbedtls_entropy_context *ctx, const char *path )
  465.    {
  466. >      int ret = MBEDTLS_ERR_ENTROPY_FILE_IO_ERROR;
  467.        FILE *f;
  468.        unsigned char buf[MBEDTLS_ENTROPY_BLOCK_SIZE];
```

FP: The initialization `int ret = MBEDTLS_ERR_ENTROPY_FILE_IO_ERROR` is used to signal correct execution of the functions `mbedtls_entropy_func` and `fwrite`. The initial value hints at this being a 'safe' return value if one were to add a conditional expression with a `goto exit` statement, but forgot to overwrite the variable `ret`.

```
library/camellia.c:559: error: DEAD_STORE
  The value written to &RK (type unsigned int*) is never used.
  557.        X[3] ^= *RK++;
  558.        X[0] ^= *RK++;
  559. >      X[1] ^= *RK++;
  560.
  561.        PUT_UINT32_BE( X[2], output,  0 );
```

FP: Pointer advancement for consistency.

```
library/ctr_drbg.c:546: error: DEAD_STORE
  The value written to &ret (type int) is never used.
  544.    int mbedtls_ctr_drbg_write_seed_file(
      mbedtls_ctr_drbg_context *ctx, const char *path )
  545.    {
  546. >      int ret = MBEDTLS_ERR_CTR_DRBG_FILE_IO_ERROR;
  547.        FILE *f;
  548.        unsigned char buf[ MBEDTLS_CTR_DRBG_MAX_INPUT ];
```

FP: Similar to the report in `library/entropy.c`. However, regarding code consistency: the else-branch in this function was not present in the other file, while they are almost identical. The else-branch:

```
else
    ret = 0;
```

```
library/entropy.c:646: error: DEAD_STORE
  The value written to &ret (type int) is never used.
  644.    int mbedtls_entropy_self_test( int verbose )
  645.    {
  646. >      int ret = 1;
  647.    #if !defined(MBEDTLS_TEST_NULL_ENTROPY)
  648.        mbedtls_entropy_context ctx;
```

FP: Return variable `ret` is assigned an exit failure value initially.

```
library/ecp_curves.c:699: error: MEMORY_LEAK
  memory dynamically allocated by call to `mbedtls_mpi_lset()`
      at line 678, column 5 is not reachable after line 699,
      column 9.
  697.    cleanup:
  698.        if( ret != 0 )
  699. >          mbedtls_ecp_group_free( grp );
  700.
  701.        return( ret );
```

FP: Line 678: `MBEDTLS_MPI_CHK( mbedtls_mpi_lset( &grp->P, 1 ) )`. This sets `grp->P->p[0]` to 1. If `ret != 0`, then the function `mbedtls_ecp_group_free` is called. This function frees the components of the ecp group, leaving only the struct to be freed. This can still be done by the caller, since it is passed via the `mbedtls_ecp_group *grp` parameter.

```
library/aes.c:910: error: DEAD_STORE
  The value written to &RK (type unsigned int*) is never used.
  908.                ( (uint32_t) FSb[ ( Y1 >> 24 ) & 0xFF ] <<
      24 );
  909.
  910. >      X3 = *RK++ ^ \
  911.                ( (uint32_t) FSb[ ( Y3       ) & 0xFF ]
           ) ^
  912.                ( (uint32_t) FSb[ ( Y0 >>  8 ) & 0xFF ] <<
       8 ) ^
```

FP: `uint32_t *RK` is a 128-bit round-key added at the end of every encryption round and before round 1. The above shows the final round key addition. The last round does not perform MixColumns. The position in the buffer is changed for consistency. One could argue this to be less justifiable in code that is very unlikely to be changed anyway.

```
library/aes.c:978: error: DEAD_STORE
  The value written to &RK (type unsigned int*) is never used.
  976.                ( (uint32_t) RSb[ ( Y3 >> 24 ) & 0xFF ] <<
      24 );
  977.
  978. >      X3 = *RK++ ^ \
  979.                ( (uint32_t) RSb[ ( Y3       ) & 0xFF ]
           ) ^
  980.                ( (uint32_t) RSb[ ( Y2 >>  8 ) & 0xFF ] <<
       8 ) ^
```

FP: The report is similar to the previous one, but for decryption (symmetry because MixColumns left out).

```
library/bignum.c:1387: error: DEAD_STORE
  The value written to &t (type unsigned long) is never used.
  1385.   #endif /* MULADDC_HUIT */
  1386.
  1387. >      t++;
  1388.
  1389.       do {
```

TP: The line `mbedtls_mpi_uint c = 0, t = 0` is used to initialize `c` and `t`. Both are used in the assembler instructions from the file `bn_mul.h`. Separate routines are used depending on whether `MULADDC_HUIT` is defined. The type definition for `mbedtls_mpi_uint` is either
`typedef uint64_t mbedtls_mpi_uint` or
`typedef uint32_t mbedtls_mpi_uint`, as specified in the file `bignum.h`. Using a post-increment on a local, non-static variable seems unnecessary.

```
library/ecp.c:1463: error: NULL_DEREFERENCE
  pointer `&l->p` last assigned on line 1458 could be null and
     is dereferenced by call to `mbedtls_mpi_fill_random()` at
     line 1463, column 9.
  1461.       do
  1462.       {
  1463. >         MBEDTLS_MPI_CHK( mbedtls_mpi_fill_random( &l,
     p_size, f_rng, p_rng ) );
  1464.
  1465.           while( mbedtls_mpi_cmp_mpi( &l, &grp->P ) >= 0
      )
```

FP: Line 1458: `mbedtls_mpi_init( &l )`. This initializes `X->p` as `NULL`, and the number of limbs is set to `0`. The struct is defined in the file `bignum.h` as:

```
typedef struct mbedtls_mpi
{
    int s;                  /*!<  integer sign      */
    size_t n;               /*!<  total # of limbs  */
    mbedtls_mpi_uint *p;    /*!<  pointer to limbs  */
} mbedtls_mpi;
```

In the function `mbedtls_mpi_fill_random`, located in the file `bignum.c`, the function call `mbedtls_mpi_grow( X, limbs )` ensures space is available to store the pseudorandom numbers generated by the following lines:
`Xp = (unsigned char*) X->p;`
`f_rng( p_rng, Xp + overhead, size );`.
Hence, the `NULL` value is never actually dereferenced.

```
library/x509_crt.c:1459: error: DEAD_STORE
  The value written to &sep (type char const *) is never used.
  1457.         CERT_TYPE( MBEDTLS_X509_NS_CERT_TYPE_SSL_CA,
                    "SSL CA" );
  1458.         CERT_TYPE( MBEDTLS_X509_NS_CERT_TYPE_EMAIL_CA,
                  "Email CA" );
  1459. >       CERT_TYPE(
      MBEDTLS_X509_NS_CERT_TYPE_OBJECT_SIGNING_CA,  "Object
      Signing CA" );
  1460.
  1461.         *size = n;
```

FP: The separator, used in the macro `PRINT_ITEM`, called from the macro `CERT_TYPE`, is overwritten from `""` to `","` for printing multiple items.

```
library/x509_crt.c:1487: error: DEAD_STORE
  The value written to &sep (type char const *) is never used.
  1485.         KEY_USAGE( MBEDTLS_X509_KU_CRL_SIGN,                "
      CRL Sign" );
  1486.         KEY_USAGE( MBEDTLS_X509_KU_ENCIPHER_ONLY,           "
      Encipher Only" );
  1487. >       KEY_USAGE( MBEDTLS_X509_KU_DECIPHER_ONLY,           "
      Decipher Only" );
  1488.
  1489.         *size = n;
```

FP: The macro `KEY_USAGE` uses the macro `PRINT_ITEM` in the same manner as last segment.

```
library/ecp.c:1820: error: MEMORY_LEAK
  memory dynamically allocated by call to `mbedtls_mpi_lset()`
     at line 1818, column 9 is not reachable after line 1820,
     column 13.
  1818.            MBEDTLS_MPI_CHK( mbedtls_mpi_lset( &R->Z, 1 )
     );
  1819.            if( f_rng != 0 )
  1820. >              MBEDTLS_MPI_CHK( ecp_randomize_jac( grp, R
     , f_rng, p_rng ) );
  1821.        }
  1822.
```

FP: The memory allocated in line 1818 comes from the function `mbedtls_mpi_grow`, called by the function `mbedtls_mpi_lset`. This enlarges to a specified number of limbs.
The call to the function `ecp_randomize_jac` calculates `Z = l * Z` for a randomized `mbedtls_mpi l` struct in the range `1 < l < p`.
Because `mbedtls_ecp_point *R` is a parameter, the allocated memory and its size are still available to the caller.

```
library/bignum.c:2247: error: MEMORY_LEAK
  memory dynamically allocated to `U1.p` by call to `
     mbedtls_mpi_lset()` at line 2187, column 5 is not reachable
      after line 2247, column 55.
  2245.    cleanup:
  2246.
  2247. >      mbedtls_mpi_free( &TA ); mbedtls_mpi_free( &TU );
     mbedtls_mpi_free( &U1 ); mbedtls_mpi_free( &U2 );
  2248.        mbedtls_mpi_free( &G ); mbedtls_mpi_free( &TB );
     mbedtls_mpi_free( &TV );
  2249.        mbedtls_mpi_free( &V1 ); mbedtls_mpi_free( &V2 );
```

FP: The components of the `mbedtls_mpi U1` are set to default values and are deallocated by the `mbedtls_mpi_free( &U1 )` function call. The struct itself is not dynamically allocated, as it is initialized using the function call `mbedtls_mpi_init( &U1 )`. This uses a reference to a local struct.

```
library/ecp.c:2181: error: NULL_DEREFERENCE
  pointer `&l->p` last assigned on line 2176 could be null and
     is dereferenced by call to `mbedtls_mpi_fill_random()` at
     line 2181, column 9.
  2179.        do
  2180.        {
  2181. >          MBEDTLS_MPI_CHK( mbedtls_mpi_fill_random( &l,
     p_size, f_rng, p_rng ) );
  2182.
  2183.            while( mbedtls_mpi_cmp_mpi( &l, &grp->P ) >= 0
      )
```

FP: The do-while loop is entered with the `mbedtls_mpi l` struct being default initialized: `mbedtls_mpi_init( &l )`.

The function `mbedtls_mpi_fill_random` incorporated the following guard before filling the struct `X` (`l` in the above segment) with `size` bytes of random:

```
if( X->n != limbs )
{
    mbedtls_mpi_free( X );
    mbedtls_mpi_init( X );
    MBEDTLS_MPI_CHK( mbedtls_mpi_grow( X, limbs ) );
}
```

Since the initialization value for `X->n` is `0`, this code will always grow the mpi. Growing the mpi is done with the `mbedtls_calloc` function, causing `&l->p` not to be null when the function `f_rng` is used to fill it.

```
library/bignum.c:2341: error: DEAD_STORE
  The value written to &i (type unsigned long) is never used.
  2339.         MBEDTLS_MPI_CHK( mbedtls_mpi_shift_r( &R, s ) );
  2340.
  2341. >       i = mbedtls_mpi_bitlen( X );
  2342.
  2343.         for( i = 0; i < rounds; i++ )
```

FP: The variable `size_t i` is immediately overwritten. Most of the other calls to the function `mbedtls_mpi_bitlen` in the same file explicitly store the result as well. Doing so may be conventional.

```
library/ecp.c:2329: error: DEAD_STORE
  The value written to &ret (type int) is never used.
  2327.                   mbedtls_ecp_restart_ctx *rs_ctx )
  2328.   {
  2329. >       int ret = MBEDTLS_ERR_ECP_BAD_INPUT_DATA;
  2330.   #if defined(MBEDTLS_ECP_INTERNAL_ALT)
  2331.         char is_grp_capable = 0;
```

FP: The variable `int ret = MBEDTLS_ERR_ECP_BAD_INPUT_DATA` is overwritten with the same value, meaning this is probably a matter of explicitly providing initial values.

Libsodium

```
src/libsodium/crypto_verify/sodium/verify.c:56: error:
    DEAD_STORE
  The value written to &v1 (type void) is never used.
  54.         }
  55.         m = _mm_movemask_epi8(_mm_cmpeq_epi32(z, zero));
  56. >       v1 = zero; v2 = zero; z = zero;
  57.
  58.         return (int) (((uint32_t) m + 1U) >> 16) - 1;
```

FP: The function returns whether two byte arrays are equal, using SSE to do 128-bit XOR- and ORs. `volatile __m128i v1, v2, z` are zeroed after usage. This report is generated once for every `zero` assignment.

```
src/libsodium/crypto_pwhash/scryptsalsa208sha256/
    pwhash_scryptsalsa208sha256.c:246: error: DEAD_STORE
  The value written to &ret (type int) is never used.
  244.        char             wanted[
      crypto_pwhash_scryptsalsa208sha256_STRBYTES];
  245.        escrypt_local_t  escrypt_local;
  246. >      int              ret = -1;
  247.
  248.        if (sodium_strnlen(str,
      crypto_pwhash_scryptsalsa208sha256_STRBYTES) !=
```

FP: The variable `int ret = -1` is initialized with an exit failure value. Similar cases are reported too.

```
src/libsodium/crypto_pwhash/argon2/argon2-core.c:499: error:
    MEMORY_LEAK
  memory dynamically allocated by call to `malloc()` at line
      493, column 10 is not reachable after line 499, column 9.
  497.        result = allocate_memory(&(instance->region),
      instance->memory_blocks);
  498.        if (ARGON2_OK != result) {
  499. >          argon2_free_instance(instance, context->flags);
  500.            return result;
  501.        }
```

FP: An `Argon2_instance_t` struct has two memory region pointers that require deallocation in case of an early exit, which are `block_region *region` and `uint64_t *pseudo_rands`.
As seen before in the other libraries, only unallocating components without directly unallocating the struct causes such a report. The memory regions are correctly zeroed, freed and have their pointers set to `NULL` by the function `argon2_free_instance`. The caller can still free the struct, as its address is passed via the parameter.

TweetNaCl

TweetNaCl does not have a single reported issue, apart from 34 reports about uninitialized values. These are all due to the expansion of the FOR macro, in which the loop counter variable is assigned `0`. The assignment does not seem to be recognized.

A.1.2 Clang Static Analyzer

BearSSL

```
src/codec/pemdec.c:418:2: warning: Out of bound memory access (
    access exceeds upper limit of memory block)
        T0_PUSH(t0_datablock[addr]);
        ^~~~~~~~~~~~~~~~~~~~~~~~~~
src/codec/pemdec.c:233:35: note: expanded from macro 'T0_PUSH'
#define T0_PUSH(v)      do { *dp = (v); dp ++; } while (0)
                                  ^~~
1 warning generated.
```

Skipped: In function `br_pem_decoder_run`. This reports was skipped because it concerns automatically generated code:
`/* Automatically generated code; do not modify directly. */`.
The code is hard to read and the report likely does not take into account function usage guidelines.

```
src/ec/ec_p256_m62.c:582:2: warning: Value stored to 'cc' is
    never read
        cc = w >> 63;
        ^    ~~~~~~~
src/ec/ec_p256_m62.c:1435:2: warning: 2nd function call argument
    is a pointer to uninitialized value
        f256_invert(zt, z[0]);
        ^~~~~~~~~~~~~~~~~~~~~
2 warnings generated.
```

1. TP: In function `f256_final_reduce`. As reported by Infer, the carry `uint64_t cc` is not propagated.

2. FP: In function `window_to_affine`. The reportedly uninitilized value: `uint64_t z[16][5]`.
   Code prior to the inversion:

   ```
   for (i = 0; (i + 1) < num; i += 2) {
       memcpy(zt, jac[i].z, sizeof zt);
       memcpy(jac[i].z, jac[i + 1].z, sizeof zt);
       memcpy(jac[i + 1].z, zt, sizeof zt);
       f256_montymul(z[i >> 1], jac[i].z, jac[i + 1].z);
   }
   if ((num & 1) != 0) {
       memcpy(z[num >> 1], jac[num - 1].z, sizeof zt);
       memcpy(jac[num - 1].z, F256_R, sizeof F256_R);
   }
   ```

   As stated by comments in the function, whenever the argument `int num` is uneven, the conditional statement ensures an extra

`F256_R` (`2^260 mod p`) is inserted in the `(num >> 1)`$^{\text{th}}$ slot of `z`. This ensures `z[0]` is initialized when `num = 1`.
For `32 >= num > 1`, the first iteration of the for-loop will put the result of `f256_montymul` in `z[0]`.
Without any deallocations before the inversion call, and the caller respecting the $(0, 32]$ window element limit, `z[0]` should always be initialized. The analyzer seems to freely pick a value for the variable `int num`, whilst this parameter is restricted to certain values.

```
src/ec/ec_p256_m64.c:1400:2: warning: 2nd function call argument
    is a pointer to uninitialized value
        f256_invert(zt, z[0]);
        ^~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `window_to_affine`. The code is similar to the code in the previous report, but for basis 2^{64} instead of 2^{52}.

```
src/ec/ecdsa_atr.c:132:2: warning: 2nd function call argument is
    a pointer to uninitialized value
        memcpy(sig, tmp, sig_len);
        ^~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `br_ecdsa_asn1_to_raw`.
Line 129: `memset(tmp, 0, sig_len)`. Additionally, integers `r` and `s` are also copied into this temporary buffer. The analyzer only acknowledges `s` being set.

```
src/ec/ecdsa_rta.c:106:14: warning: Out of bound memory access (
    access exceeds upper limit of memory block)
        tmp[off ++] = 0x02;
        ~~~~~~~~~~~~~~~~~~
src/ec/ecdsa_rta.c:106:14: warning: Out of bound memory access (
    accessed memory precedes memory block)
        tmp[off ++] = 0x02;
        ~~~~~~~~~~~~~~~~~~
2 warnings generated.
```

FP: In function `br_ecdsa_raw_to_asn1`. The preceding conditional branches apply either `off = 3` or `off = 2` as the variable `off`'s initial value. Next, we may assume $rlen <= 125$ after the following code segment:

```
if (rlen > 125 || slen > 125) {
    return 0;
}
```

This is followed by two `off++` post-increments and `off += rlen`.
`off` is in the range $2 <= off <= 3 + rlen + 2 <= 130$.
The size of `unsigned char tmp[257]` is 257 bytes.

The warning is odd, as it implies the same line accesses memory proceeding- and exceeding the upper limit of the memory block. This would be less strange if the variable `off` took on the value of a freely chosen parameter, or if `off` was positioned in a loop with a condition depending on a freely chosen parameter. Neither of these are the case.

```
src/rsa/rsa_oaep_pad.c:90:23: warning: Out of bound memory
    access (access exceeds upper limit of memory block)
        buf[k - src_len - 1] = 0x01;
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `br_rsa_oaep_pad`.
On line 70, a conditional statement causes an early exit with the disjunction operands being:

Disjunction operands	Assumption afterwards
`k < ((hlen << 1) + 2)`	`(hlen << 1) + 1 <= k - 1`
`src_len > (k - (hlen << 1) - 2)`	`(hlen << 1) + 1 <= k - src_len - 1`
`dst_max_len < k`	`k <= dst_max_len`

Since `buf = dst` and the arguments `void *dst, size_t dst_max_len` may be assumed well-chosen, we know
$k - src\_len - 1 < k <= dst\_max\_len$.
The upper limit of memory should therefore not be accessible under these conditions.

```
src/symcipher/aes_x86ni_ctr.c:71:8: warning: Pointer arithmetic
    on non-array variables relies on memory layout, which is
    dangerous
                x0 = _mm_insert_epi32(ivx, br_bswap32(cc + 0),
                    3);
                     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/usr/lib/llvm-6.0/lib/clang/6.0.0/include/smmintrin.h:990:39:
    note: expanded from macro '_mm_insert_epi32'
                                       __a[(N) & 3] = (I);
                                                            \
                                       ^~~~~~~~~~~~
src/symcipher/aes_x86ni_ctr.c:72:8: warning: Pointer arithmetic
    on non-array variables relies on memory layout, which is
    dangerous
                x1 = _mm_insert_epi32(ivx, br_bswap32(cc + 1),
                    3);
                     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/usr/lib/llvm-6.0/lib/clang/6.0.0/include/smmintrin.h:990:39:
    note: expanded from macro '_mm_insert_epi32'
                                       __a[(N) & 3] = (I);
                                                            \
                                       ^~~~~~~~~~~~
src/symcipher/aes_x86ni_ctr.c:73:8: warning: Pointer arithmetic
    on non-array variables relies on memory layout, which is
    dangerous
                x2 = _mm_insert_epi32(ivx, br_bswap32(cc + 2),
                    3);
                     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/usr/lib/llvm-6.0/lib/clang/6.0.0/include/smmintrin.h:990:39:
    note: expanded from macro '_mm_insert_epi32'
                                       __a[(N) & 3] = (I);
                                                            \
                                       ^~~~~~~~~~~~
src/symcipher/aes_x86ni_ctr.c:74:8: warning: Pointer arithmetic
    on non-array variables relies on memory layout, which is
    dangerous
                x3 = _mm_insert_epi32(ivx, br_bswap32(cc + 3),
                    3);
                     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/usr/lib/llvm-6.0/lib/clang/6.0.0/include/smmintrin.h:990:39:
    note: expanded from macro '_mm_insert_epi32'
                                       __a[(N) & 3] = (I);
                                                            \
                                       ^~~~~~~~~~~~
4 warnings generated.
```

FP: The macro, which is NOT part of the BearSSL library:

```
#define _mm_insert_epi32(X, I, N) (__extension__      \
    ({ __v4si __a = (__v4si)(__m128i)(X);             \
    __a[(N) & 3] = (I);                               \
    (__m128i)__a;}))
```

The only pointer arithmetic is the addition of `(N) & 3` to the base address `__a`. The typedef for the cast of `__a` can be found in `xmmintrin.h` (by following the include chain):
`typedef int __v4si __attribute__((__vector_size__(16)));`.
The Clang Static Analyzer does not seem to pick up on the array of (usually) four 32-bit integers, but treats it as a non-array variable.

```
In file included from src/symcipher/chacha20_sse2.c:26:
In file included from src/inner.h:2527:
In file included from /usr/lib/llvm-6.0/lib/clang/6.0.0/include/
    x86intrin.h:29:
In file included from /usr/lib/llvm-6.0/lib/clang/6.0.0/include/
    immintrin.h:32:
In file included from /usr/lib/llvm-6.0/lib/clang/6.0.0/include/
    xmmintrin.h:2969:
/usr/lib/llvm-6.0/lib/clang/6.0.0/include/emmintrin.h:4312:26:
    warning: Pointer arithmetic on non-array variables relies on
    memory layout, which is dangerous
  return (unsigned short)__b[__imm & 7];
                         ^~~~~~~~~~~~~~

1 warning generated.
```

FP: `__b` is defined as: `__v8hi __b = (__v8hi)__a` with the typedef:
`typedef short __v8hi __attribute__((__vector_size__(16)))`.
This is almost the same issue as last report. Again, these are reports regarding llvm code rather than BearSSL code.

```
src/x509/skey_decoder.c:504:13: warning: Out of bound memory
    access (access exceeds upper limit of memory block)
        if (len == a2[0]) {
                   ^~~~~

1 warning generated.
```

Skipped (automatically generated code):
In function `br_skey_decoder_run`.
Line 500: `const unsigned char *a2 = &t0_datablock[T0_POP()]`.
The macro `T0_POP` is defined as `#define T0_POP() (*-- dp)` and `t0_datablock` is a constant byte-array filled with 40 bytes. The code is located inside a repeated switch, stopping only when switch variable `uint32_t t0x`, assigned `t0x = T0_NEXT(&ip)`, takes on the value `0`, the `ret` instruction. Due to the size of the function, it is very hard to judge the correctness of the report.

```
src/x509/x509_decoder.c:479:13: warning: Division by zero
        T0_PUSHi(a % b);
                 ~~~~^~~
src/x509/x509_decoder.c:389:47: note: expanded from macro '
    T0_PUSHi'
#define T0_PUSHi(v)     do { *(int32_t *)dp = (v); dp ++; } while
     (0)
                                                    ^
src/x509/x509_decoder.c:520:13: warning: Division by zero
        T0_PUSHi(a / b);
                 ~~~~^~~
src/x509/x509_decoder.c:389:47: note: expanded from macro '
    T0_PUSHi'
#define T0_PUSHi(v)     do { *(int32_t *)dp = (v); dp ++; } while
     (0)
                                                    ^
src/x509/x509_decoder.c:639:2: warning: Out of bound memory
    access (access exceeds upper limit of memory block)
        T0_PUSH(t0_datablock[addr]);
        ^~~~~~~~~~~~~~~~~~~~~~~~~~
src/x509/x509_decoder.c:388:35: note: expanded from macro '
    T0_PUSH'
#define T0_PUSH(v)      do { *dp = (v); dp ++; } while (0)
                                   ^~~
src/x509/x509_decoder.c:660:13: warning: Out of bound memory
    access (access exceeds upper limit of memory block)
        if (len == a2[0]) {
                   ^~~~~
4 warnings generated.
```

- 1. & 3. Skipped (again, automatically generated).
  In function `br_x509_decoder_run`. The first division by zero is labeled the `/* %25 */` instruction, whilst the second is normal division (`/* / */`). It is probable that the provided `void *t0ctx` argument needs to ensure the second operands are not `0`. The Clang Static Analyzer does not know about usage guidelines like this.

```
src/x509/x509_knownkey.c:89:7: warning: Casting a non-structure
    type to a structure type and accessing a field can lead to
    memory access errors or data corruption
        xc = (const br_x509_knownkey_context *)ctx;
              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `kk_get_pkey`.
`const br_x509_class *const *ctx` is converted to type
`const br_x509_knownkey_context *`. First of all, scan-tool does not seem to acknowledge the fact that both cast types are pointers, meaning the memory layout is only reinterpreted. This should never cause data corrup-

tion by itself. The structure that is pointed to in the cast type, found in `bearssl_x509.h`:

```
typedef struct {
    /** \brief Reference to the context vtable. */
    const br_x509_class *vtable;
#ifndef BR_DOXYGEN_IGNORE
    br_x509_pkey pkey;
    unsigned usages;
#endif
} br_x509_knownkey_context;
```

The cast lowers the pointer from a
<*pointer to a <constant pointer to a constant* `br_x509_class` struct>>, to a <*constant pointer to a* `br_x509_knownkey_context` struct>.
The pointer `const br_x509_class *vtable` is the first member, causing it to correctly line up with the cast. The caller needs to ensure the other members also line up correctly. This causes the `br_x509_pkey pkey` struct member to be returnable by reference. If this is done carefully, the cast should not result in problems. This type of pointer casting is common in object oriented programming in C.

```
tools/certs.c:136:2: warning: Null pointer passed as an argument
     to a 'nonnull' parameter
        VEC_ADDMANY(*dst, &VEC_ELT(tas, 0), num);
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

tools/brssl.h:126:3: note: expanded from macro 'VEC_ADDMANY'
                memcpy((vec).buf + (vec).ptr, \
                ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~

tools/certs.c:166:8: warning: Casting a non-structure type to a
    structure type and accessing a field can lead to memory
    access errors or data corruption
        xwc = (x509_noanchor_context *)ctx;
              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~

===[4 more reports of the same type left out]===

tools/certs.c:216:8: warning: Casting a non-structure type to a
    structure type and accessing a field can lead to memory
    access errors or data corruption
        xwc = (x509_noanchor_context *)ctx;
              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~

7 warnings generated.
```

- 1: FP: In function `read_trust_anchors`.
  `anchor_list *dst` is a parameter. The macro definition without its body is `VEC_ADDMANY(vec, xp, num)`. The report is another instance of the case where a function like `memcpy` can be called with a `null`

argument and a `0` argument that specifies the length. The first report by Infer on BearSSL is similar.

- 2-7: FP: More instances of the pointer casting report.

```
tools/skey.c:483:2: warning: Value stored to 'ret' is never read
        ret = 1;
        ^     ~

1 warning generated.
```

FP: In function `decode_key`. The value assigned to the variable `ret` will be overwritten in any switch case.

mbed TLS

```
aes.c:1256:31: warning: The right operand of '^' is a garbage
    value
            tmp[i] = input[i] ^ t[i];
                              ^ ~~~~

1 warning generated.
```

FP: In function: `mbedtls_aes_crypt_xts`.

```
unsigned char *t =
    mode == MBEDTLS_AES_DECRYPT ? prev_tweak : tweak
```

This assigns the buffer to be the previous tweak for decryption with leftover bytes. The leftover bytes are the start of the ciphertext before it, meaning the previous tweak is needed for decryption (XOR'd). `tweak_prev` is conditionally defined, whenever the following holds:
`leftover && ( mode == MBEDTLS_AES_DECRYPT ) && blocks == 0`
(where blocks is the amount of full blocks left). `tweak` is updated every block before handling the leftover: `mbedtls_gf128mul_x_ble( tweak, tweak )`. `t` could contain a garbage value if the function would run without a full block, but this is prevented by a condition in one of the first lines: `length < 16`. Therefore, the XOR should not be having a right operand garbage value.

```
asn1write.c:49:17: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
        *--(*p) = (unsigned char) len;
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

===[Similar entries left out]===
asn1write.c:243:13: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
    *--(*p) = val;
    ~~~~~~~~~~~~~

asn1write.c:245:22: warning: This statement is never executed
    if( val > 0 && **p & 0x80 )
                   ^

asn1write.c:248:21: warning: This statement is never executed
            return( MBEDTLS_ERR_ASN1_BUF_TOO_SMALL );
                    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

../include/mbedtls/asn1.h:57:60: note: expanded from macro '
    MBEDTLS_ERR_ASN1_BUF_TOO_SMALL'
#define MBEDTLS_ERR_ASN1_BUF_TOO_SMALL                    -0
    x006C  /**< Buffer too small when writing ASN.1 data
    structure. */
                                                           ^~~~~~

===[Two more errors like the first one left out]===
13 warnings generated.
```

1. FP: In function `mbedtls_asn1_write_len`. The function is meant to write a length field to a buffer in ASN.1 format. Doing so involves specifying the length of the length field, and its value, inside the buffer *pointed to* by the location the *pointer* `unsigned char **p` *is pointing to*.
   The pointer `unsigned char *start` points to the start of the buffer. Before any writing of $1 <= x <= 5$ bytes is done (a 32-bit integer and one byte), not passing the check `*p - start < x` causes an early exit. This counters out of bounds memory access, assuming the arguments are valid.

3. FP: In function `mbedtls_asn1_write_int`. Before the conditional, the assignment `*--(*p) = val` is done. Both are parameters: `unsigned char **p, unsigned char *start, int val`.
   The claim that `**p` would never be executed stems from the first operand of the `&&` operator. This would imply $val <= 0$ all the time, which should not be the case.

4. FP: The statement is located in the then-branch of the conditional in (3), guarded by a length check. If the right operand in (3) is though to be unreachable, this implies the conditional is always considered false (because of the `&&` operator). Extending the same (incorrect) logic, the statement in (4) will never execute either.

```
bignum.c:721:15: warning: Out of bound memory access (accessed
    memory precedes memory block)
    s[slen++] = '\n';
    ~~~~~~~~~~~~~~~~
bignum.c:1160:15: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
        z = ( *d <  c );     *d -= c;
              ^~
bignum.c:1166:15: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
        z = ( *d < c ); *d -= c;
              ^~
bignum.c:2341:5: warning: Value stored to 'i' is never read
    i = mbedtls_mpi_bitlen( X );
    ^   ~~~~~~~~~~~~~~~~~~~~~~
4 warnings generated.
```

FP: In function `mbedtls_mpi_write_file`. The `char` buffer initialized as `char s[ MBEDTLS_MPI_RW_BUFFER_SIZE ]` is indexed using a post-incremented version of the variable `slen = strlen( s )`. The macro determining the size of the buffer incorporates extra bytes (for "`the newline * characters and the '\0'`"), but the `strlen` function counts until the `\0` (excluded).

```
chacha20.c:555:22: warning: 1st function call argument is a
    pointer to uninitialized value
        ASSERT( 0 == memcmp( output, test_output[i],
            test_lengths[i] ),
^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
chacha20.c:525:17: note: expanded from macro 'ASSERT'
        if( ! ( cond ) )                \
              ^~~~
1 warning generated.
```

FP: In function `mbedtls_chacha20_self_test`. `output` is an array containing the result of the function `mbedtls_chacha20_crypt` with test parameters. This function uses the function `mbedtls_chacha20_update`, which initializes the `output` array if it was not initialized already.

```
cipher.c:1018:17: warning: This statement is never executed
        return( ret );
                ^~~
1 warning generated.
```

FP: In function `mbedtls_cipher_crypt`. The statement is conditionally executed whenever `ret = mbedtls_cipher_reset( ctx ) ) != 0` evaluates to true.
`mbedtls_cipher_reset` can be non-zero if `ctx->cipher_info == NULL` (pointer argument with same name).
This check is actually unneeded, as the original function already did the following check: `CIPHER_VALIDATE_RET( ctx != NULL )`. In the meantime,

only the members of `ctx->cipher_info` have changed, but it it not set to `NULL`. With this type of code, this could still be considered a feasible check for safety.

```
ecdsa.c:300:13: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
        if( *p_sign_tries++ > 10 )
            ^~~~~~~~~~~~~~~
ecdsa.c:300:14: warning: Pointer arithmetic on non-array
    variables relies on memory layout, which is dangerous
        if( *p_sign_tries++ > 10 )
             ^~~~~~~~~~~~~~
ecdsa.c:313:17: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
            if( *p_key_tries++ > 10 )
                ^~~~~~~~~~~~~~
ecdsa.c:313:18: warning: Pointer arithmetic on non-array
    variables relies on memory layout, which is dangerous
            if( *p_key_tries++ > 10 )
                 ^~~~~~~~~~~~~
4 warnings generated.
```

TP: In function `ecdsa_sign_restartable`. The pointer `p_sign_tries = &rs_ctx->sig->sign_tries` is within bounds in the first do-while iteration. However, the order of the left operand of the `>` operator seems to be wrong. `*p_sign_tries++` increments the address that `p_sign_tries` points to for the next iteration of the do-while loop; it then dereferences the pointer. The author likely meant to do `(*p_sign_tries)++`, in which case it acts as a counter. `(*p_sign_tries)++` would increment `*p_sign_tries` every do-while iteration.

```
gcm.c:825:18: warning: 1st function call argument is a pointer
    to uninitialized value
            if ( memcmp( buf, ct[j * 6 + i], pt_len[i] ) != 0 ||
                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `mbedtls_gcm_self_test`. `unsigned char buf[64]` is initialized with a call to the function `mbedtls_gcm_crypt_and_tag`, calling the function `mbedtls_gcm_update` which treats `buf` as its output.

```
hkdf.c:153:15: warning: 2nd function call argument is a pointer
    to uninitialized value
        ret = mbedtls_md_hmac_update( &ctx, t, t_len );
              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `mbedtls_hkdf_expand`. The function ends up calling

```
/** Digest update function */
int (*update\_func)( void *ctx,
                     const unsigned char *input,
                     size\_t ilen );
```

with `unsigned char t[MBEDTLS_MD_MAX_SIZE]` as the `input` argument and its size as the `ilen` argument. `update` is a function pointer for an opaque struct. The following calls are used to set up `ctx`, which determines the update function, called from line 153:

```
mbedtls_md_init( &ctx );
if( (ret = mbedtls_md_setup( &ctx, md, 1) )
```

This means that the update function, which is supplied as the first argument in line 153, is dependent on supplying the right
`const mbedtls_md_info_t *md` argument to the function. If this `md` supplies an update function supporting/overwriting uninitialized buffers, there is no problem.

```
pem.c:473:9: warning: Null pointer passed as an argument to a '
    nonnull' parameter
        memcpy( p, c, len );
        ^~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `mbedtls_pem_write_buffer`.
The assignment `c = encode_buf` is used, but if `encode_buf` were `NULL`, this would cause an early exit by the following conditional:

```
if( ( ret = mbedtls_base64_encode( encode_buf, use_len,
    &use_len, der_data, der_len ) ) != 0 )
```

This returns the non-zero
`return( MBEDTLS_ERR_BASE64_BUFFER_TOO_SMALL )` in case the destination (`encode_buf`) is `NULL`.

```
pkcs12.c:205:17: warning: 2nd function call argument is a
    pointer to uninitialized value
    if( ( ret = mbedtls_cipher_setkey( &cipher_ctx, key, 8 *
        keylen, (mbedtls_operation_t) mode ) ) != 0 )
                ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pkcs12.c:208:17: warning: 2nd function call argument is a
    pointer to uninitialized value
    if( ( ret = mbedtls_cipher_set_iv( &cipher_ctx, iv,
        cipher_info->iv_size ) ) != 0 )
                ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2 warnings generated.
```

FP: In function `mbedtls_pkcs12_pbe`. Again, the arguments are the buffers to be filled, which may be uninitialized.

```
pkcs5.c:386:13: warning: 2nd function call argument is a pointer
     to uninitialized value
            memcmp( result_key[i], key, key_len[i] ) != 0 )
            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `mbedtls_pkcs5_self_test`, a test function. `key` is the output of `mbedtls_pkcs5_pbkdf2_hmac`, one of the functions that is being tested.

```
pkwrite.c:227:10: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
    *--c = 0;
     ~~~~~~~~
pkwrite.c:370:14: warning: Out of bound memory access (access
    exceeds upper limit of memory block)
        *--c = 0;
         ~~~~~~~~
2 warnings generated.
```

FP: In function `mbedtls_pk_write_pubkey_der`. Checks are in place to prevent going out of bounds: the assignment `c = buf + size` is followed by the segment:

```
if( c - buf < 1 )
    return( MBEDTLS_ERR_ASN1_BUF_TOO_SMALL );
```

```
rsa.c:1828:5: warning: 2nd function call argument is a pointer
    to uninitialized value
    memcpy( p, salt, slen );
    ^~~~~~~~~~~~~~~~~~~~~~
rsa.c:1986:10: warning: Out of bound memory access (accessed
    memory precedes memory block)
    *p++ = (unsigned char)( 0x08 + oid_size + hashlen );
    ~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
rsa.c:2208:16: warning: The left operand of '>>' is a garbage
    value
    if( buf[0] >> ( 8 - siglen * 8 + msb ) )
        ~~~~~~ ^
3 warnings generated.
```

1. FP: In function `mbedtls_rsa_rsassa_pss_sign`. This is initialized in:

   ```
   /* Generate salt of length slen */
   if( ( ret = f_rng( p_rng, salt, slen ) ) != 0 )
   ```

2. FP: In function `rsa_rsassa_pkcs1_v15_encode`.
   The pointer `unsigned char *p = dst` has a length passed by the `size_t dst_len` parameter. This is saved in the assignment `size_t nb_pad = dst_len`, and checked before writing to `p`.

3. FP: In function `mbedtls_rsa_rsassa_pss_verify_ext`.
   `unsigned char buf[MBEDTLS_MPI_MAX_SIZE]` is overwritten in any of the two ternary operator expressions in:

   ```
   ret = ( mode == MBEDTLS_RSA_PUBLIC )
       ? mbedtls_rsa_public(  ctx, sig, buf )
       : mbedtls_rsa_private( ctx, f_rng, p_rng, sig, buf );
   ```

   This report seems to happen when an array size is specified by a macro, such as `#define MBEDTLS_MPI_MAX_SIZE 1024` (defined in `bignum.h`).

```
timing.c:442:21: warning: This statement is never executed
        millisecs = mbedtls_timing_get_timer( &hires, 0 );
                    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
timing.c:494:13: warning: This statement is never executed
        if( verbose != 0 )
            ^~~~~~~
timing.c:510:18: warning: This statement is never executed
        cycles = mbedtls_timing_hardclock();
                 ^~~~~~~~~~~~~~~~~~~~~~~~~~
3 warnings generated.
```

The second and third explanations for the above reports are hand-wavy. One cannot reasonably expect a static analyzer to handle these cases correctly. These reports are merely included for completeness.
In function `mbedtls_timing_self_test`:

1. FP: The previous statements:

   ```
   mbedtls_set_alarm( (int) secs );
   while( !mbedtls_timing_alarmed )
       ;
   ```

   The condition is based on a volatile integer:
   `volatile int mbedtls_timing_alarmed = 0`, thus the value may change at any time. This could be an oversight in the way volatile integers are interpreted.

2. Skipped: This code is accessible, but only after a `goto` statement has been taken. The variable `int hardfail = 0` has not changed before the conditional guarding this code is reached, which is

   ```
   hard_test:
   if( hardfail > 1 )
   ```

   Afterwards, the following code segment can actually reach the `hard_test` label:

   ```
   hardfail++;
   goto hard_test;
   ```

   This should only happen in when the cycle counter wraps twice during the test. A detailed explanation on this behavior is provided in the source code.

3. Skipped: The Clang Static Analyzer making abstractions (for this rather specialized function/test) seems like the most plausible explanation.

```
x509_create.c:172:17: warning: 4th function call argument is a
    pointer to uninitialized value
                mbedtls_asn1_store_named_data( head, oid, strlen
                    ( oid ),
                ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
x509_create.c:311:13: warning: Out of bound memory access (
    access exceeds upper limit of memory block)
    *--(*p) = 0;
    ~~~~~~~~~~~
2 warnings generated.
```

1. FP: In function `mbedtls_x509_string_to_names`. The entire statement:

   ```
   mbedtls_asn1_named_data* cur =
   mbedtls_asn1_store_named_data( head, oid, strlen( oid ),
   (unsigned char *) data,
   d - data );
   ```

   `data` contains data that may be stored, `d - data` is the amount of data that will be stored. The difference between `d` and `data` is initially `0`, as `char *d = data`. This is safely expanded in later iterations of the outer loop (`while( c <= end )`) by the assignment `*(d++) = *c`.

2. FP: In function `mbedtls_x509_write_sig`. Again, a statement guarded by length checks.

```
aes/crypt_and_hash.c:452:13: warning: 2nd function call argument
    is a pointer to uninitialized value
            mbedtls_md_update( &md_ctx, key, keylen );
            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `main`. The secret key is read from the command line or from a file, both of which can be passed via `argv[6]`. If `argc != 7`, the program will exit.

```
hash/generic_sum.c:227:9: warning: Value stored to 'ret' is
    never read
        ret |= generic_check( md_info, argv[3] );
        ^     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `main`. The initialization `int ret = 1` is solely used to signify a couple of functions have return values (which may be used in the future), but is never read or returned.

```
pkey/pk_decrypt.c:141:17: warning: 2nd function call argument is
    a pointer to uninitialized value
    if( ( ret = mbedtls_pk_decrypt( &pk, buf, i, result, &olen,
        sizeof(result),
                ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `main`. `unsigned char buf[512]` is the buffer read from a text file, containing the RSA data that needs to be decrypted. Its length, stored in the variable `size_t i`, is incremented for every byte written to `buf`, meaning everything within this size is initialized.

```
pkey/rsa_verify.c:144:17: warning: 8th function call argument is
    a pointer to uninitialized value
    if( ( ret = mbedtls_rsa_pkcs1_verify( &rsa, NULL, NULL,
        MBEDTLS_RSA_PUBLIC,
                ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

FP: In function `main`. Same scenario as the last warning.

Libsodium

```
crypto_generichash/blake2b/ref/blake2b-ref.c:120:20: warning: 1
    st function call argument is a pointer to uninitialized value
        S->h[i] ^= LOAD64_LE(p + sizeof(S->h[i]) * i);
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
./include/sodium/private/common.h:61:24: note: expanded from
    macro 'LOAD64_LE'
#define LOAD64_LE(SRC) load64_le(SRC)
                       ^~~~~~~~~~~~~~
crypto_generichash/blake2b/ref/blake2b-ref.c:358:13: warning:
    This statement is never executed
            sodium_misuse();
            ^~~~~~~~~~~~~~~
crypto_generichash/blake2b/ref/blake2b-ref.c:362:13: warning:
    This statement is never executed
            sodium_misuse();
            ^~~~~~~~~~~~~~~
crypto_generichash/blake2b/ref/blake2b-ref.c:397:13: warning:
    This statement is never executed
            sodium_misuse();
            ^~~~~~~~~~~~~~~
crypto_generichash/blake2b/ref/blake2b-ref.c:401:13: warning:
    This statement is never executed
            sodium_misuse();
            ^~~~~~~~~~~~~~~
5 warnings generated.
```

1. FP: In function `blake2b_init_param`. Part of the operands are parameters (these are `blake2b_state *S, const blake2b_param *P`

where `p = (const uint8_t *) (P)`) and `size_t i` is the loop counter variable starting from `0`.

2. - 5. (same scenario)
   FP: In function `blake2b`. A helper function has its return value checked for values smaller than `0` in a conditional statement, with `sodium_misuse()` as its then-branch. The helper function either returns `0` signaling normal execution, or it calls the function `sodium_misuse` itself (causing the program to quit).

```
crypto_hash/sha256/cp/hash_sha256_cp.c:162:31: warning: Out of
    bound memory access (accessed memory precedes memory block)
            state->buf[r + i] = PAD[i];
            ~~~~~~~~~~~~~~~~~~~~~~^~~~

crypto_hash/sha256/cp/hash_sha256_cp.c:166:31: warning: Out of
    bound memory access (accessed memory precedes memory block)
            state->buf[r + i] = PAD[i];
            ~~~~~~~~~~~~~~~~~~~~~~^~~~

2 warnings generated.
```

FP: In function `SHA256_Pad`.
`r = (unsigned int) ((state->count >> 3) & 0x3f)`, so
$0 <= r <= 63$. The struct `crypto_hash_sha256_state` has the member `uint8_t buf[64]`. The first warning is in the then-branch, the second in the else branch, which contain similar code.

1. If `r < 56`, `i` will be in $0 <= i < 56 - r <= 56$, so that $r + i < r + (56 - r) = 56$. This will not go out of range.
2. If `!(r < 56)`, then $56 <= r <= 63$. Then `i` will be in $0 <= i < 64 - r <= 64$, so that $r + i < r + (64 - r) = 64$. This will not go out of range.

Both of the reports claim memory preceding the memory block is accessed, but since `56 - r` and `64-r` will always be at least `1`, there is no way for `i` to be treated as a negative value in the index expression `r + i`.

```
crypto_onetimeauth/poly1305/donna/poly1305_donna.c:52:42:
    warning: Out of bound memory access (accessed memory precedes
     memory block)
            st->buffer[st->leftover + i] = m[i];
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~

1 warning generated.
```

FP: In function `poly1305_state_internal_t`. The struct associated with the parameter `poly1305_state_internal_t *st` has two relevant members:

```
unsigned long long leftover;
unsigned char buffer[poly1305_block_size];
```

The function has three parts:

1. Handle leftover stored in the Poly1305 state. If the previous leftover and the new amount of bytes is smaller than `#define poly1305_block_size 16` (from the file `poly1305_donna64.h`), return without calculating a MAC.
2. Handle a total of `(bytes & ~(poly1305_block_size - 1))` full blocks.
3. Store any leftover, which should be less than 16 bytes.

Since `buffer` is of size `poly_1305_block_size`, and leftover is reduced to `0` in the first stage, storing up to `poly_1305_block_size - 1` bytes in `buffer` will not go out of bounds. The variables `i` and `st->leftover` can not be negative, ruling out the access of memory preceding the block.

```
crypto_pwhash/argon2/argon2-core.c:102:28: warning: The code
    calls sizeof() on a pointer type. This can produce an
    unexpected result
    memcpy(&memory, &base, sizeof memory);
                           ^      ~~~~~~

crypto_pwhash/argon2/argon2-core.c:493:34: warning: the
    computation of the size of the memory allocation may overflow
          malloc(sizeof(uint64_t) * instance->segment_length)) ==
              NULL) {
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

2 warnings generated.
```

1. FP: In function `allocate_memory`. The `sizeof` of a pointer is used to determine the size needed to store the address of `base`.
2. FP: In function `argon2_initialize`. The caller needs to ensure that the variable `uint32_t segment_length` in `typedef struct Argon2_instance_t` is not too large.

```
crypto_pwhash/crypto_pwhash.c:171:13: warning: This statement is
     never executed
    return -1;
           ^

1 warning generated.
```

FP: In function `crypto_pwhash_str_alg`. This is expected:

```
sodium_misuse();
/* NOTREACHED */
return -1;
```

```
sodium/core.c:35:21: warning: This statement is never executed
            return -1; /* LCOV_EXCL_LINE */
                   ^
sodium/core.c:50:17: warning: This statement is never executed
        return -1; /* LCOV_EXCL_LINE */
               ^
sodium/core.c:211:17: warning: This statement is never executed
        return -1; /* LCOV_EXCL_LINE */
               ^
3 warnings generated.
```

FP: These depend on integers marked volatile:

```
static volatile int initialized;
static volatile int locked;
```

The Clang Static Analyzer does not always seem to treat these right.

```
crypto_pwhash/scryptsalsa208sha256/crypto_scrypt-common.c:42:18:
    warning: Pointer arithmetic on non-array variables relies on
    memory layout, which is dangerous
        *dst++ = itoa64[src & 0x3f];
                 ^~~~~~~~~~~~~~~~~~
crypto_pwhash/scryptsalsa208sha256/crypto_scrypt-common.c
    :219:14: warning: Pointer arithmetic on non-array variables
    relies on memory layout, which is dangerous
    *dst++ = itoa64[N_log2];
             ^~~~~~~~~~~~~
2 warnings generated.
```

FP: This is a pointer, and the pointer arithmetic should be alright.

```
static const char *const itoa64 =
"./0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
```

This seems like an oversight in the Clang Static Analyzer, as this is common practice. In the second warning, `itoa64[N_log2]` is functionally equivalent to `*(itoa64 + N_log2)`.

```
[A nonalarming libsodium warning was removed here because it
    contains unicode characters]
crypto_onetimeauth/poly1305/sse2/poly1305_sse2.c:813:29: warning
    : Out of bound memory access (access exceeds upper limit of
    memory block)
            final[leftover] = 1;
            ~~~~~~~~~~~~~~~~^~~
1 warning generated.
```

FP: In function `poly1305_finish_ext`. The array `CRYPTO_ALIGN(16) unsigned char final[32] = { 0 }` has a size of 32 bytes. The caller must ensure the argument passed as `unsigned long long leftover` is smaller than 32, as this is meant to be

the final block. This is done by the
`poly1305_update` function, which stores any leftover bytes after processing full blocks.

TweetNaCl

The Clang Static Analyzer did produce any warnings whilst analyzing TweetNaCl.

A.1.3 Splint

BearSSL

```
inc/bearssl_hash.h:1166:27: Storage *ctx reachable from
    parameter contains 1 undefined field: impl
Storage derivable from a parameter, return value or global is
    not defined.
Use /*@out@*/ to denote passed or returned storage which need
    not be defined.
```

FP: The reports suggest to use an `/*@out@*/` annotation. The Splint manual [12] covers an example usage of this annotation for a setter. The `out` annotation is used for "a pointer to storage that may be undefined". This is not useful for a getter function like in BearSSL.
The check is useful, given one inserts annotations used by Splint that express the intent of the programmer. From a Splint perspective, it could be interesting to try to adapt detection to one or more C coding styles. For example, the Google C style guide [37] provides guidelines on where to place input and output parameters.

As the output produced by Splint is repetitive and long, here follows a quick summary of the other report types:

1. Many reports are due to the Splint specific annotations missing:
   (a) `/*@out@*/` for pointers possibly pointing to undefined storage.
   (b) `/*@null@*/` for parameters that may be null.
   (c) `/*@only@*/` to "indicate a reference is the only pointer to the object it points to" [34], thus it also needs to be freed from this reference.
2. Integers directly used as booleans.
3. Implicit type conversions.

mbed TLS

Firstly, with Splint, running `splint filename.c` with the `-preproc` flag is necessary. Without this flag, a simple program such as the following will error, by treating the `CHAR_BIT` definition from `limits.h` as `0`:

```
1 #include <limits.h>
2 #include <stdio.h>
3 int main(int /*@unused@*/ argc, char /*@unused@*/ *argv[]){
4         printf("%d\n", CHAR_BIT);
5         #if (CHAR_BIT == 0)
6         #error (CHAR_BIT == 0)
7         #endif
8         return 0;
9 }
```

With the flag, no warnings are produced.
The flag is only needed for mbed TLS, as it checks limits in `check_config.h`, included from `config.h`.

```
include/mbedtls/cipher.h:430:19: Statement has no effect: do { {
    } } } whi...
Statement has no visible effect -- no values are modified.
```

FP: This is the alternative definition of the `MBEDTLS_INTERNAL_VALIDATE_RET` macro, which is meant to have no effect. The definition without effect is used when `MBEDTLS_CHECK_PARAMS` is not defined, protecting it from external usage.

```
library/ccm.c:200:13: Left operand of << may be negative (
    boolean):
(add_len > 0) << 6
The left operand to a shift operator may be negative (behavior
    is
implementation-defined).
```

The relational operators will only ever return `0` or `1`, meaning the left operand of the left shift can not be negative [18], as defined in section 6.5.8 of the C11 draft.

```
library/ccm.c:206:43: Operand of ++ is non-numeric (unsigned
    char): i
library/ccm.c:207:11: Incompatible types for - (int, unsigned
    char): 15 - i
```

Reports such as the above generate a lot of noise in the output of Splint, at least while dealing with cryptography code.

1. FP: Unsigned char is a common way of representing bytes, for which a post-increment can be feasible.

2. FP: The types are dealt with by promoting the unsigned char to an integer, to then subtract this from the integer `15`. As the unsigned char stays within the positive range of signed integers, no problems occur.

Splint offers several flags to deal with less of these reports, for instance:

1. The `match-any-integral` flag is used to make an arbitrary integral type match any integral type. This flag causes no warnings to be generated whilst running Splint on the code in Figure A.1, although there are more specific rules that can do the same.

2. Several flags for setting the types of `/*@integraltype@*/`, `/*@unsignedintegraltype@*/` and `/*@signedintegraltype@*/` to (unsigned) long. An example is the flag `long-unsigned-unsigned-integral`, which is more specific than the `match-any-integral`, but effective for silencing the warning shown in Figure A.2 on the next page.

3. Flags such as `charint --- char and int are equivalent`.
One of the consequences of using this flag is that post-incrementing a character will not generate a warning.

Other report types that occur frequently:

1. `Array fetch using non-integer, unsigned char`.

2. `Incompatible types for - (unsigned char, int)` (other variations occur for different integer types).

3. `Assignment of int to unsigned char` (multiple variations).

The frequency of these reports can be reduced if one uses a carefully chosen set of flags, in line with the content of the file that is analyzed. Secondly, some of the reports could be avoided if Splint looked at (constant) variable ranges when a type conversion happens.

```
1 #include <stddef.h>
2 int main(int /*@unused@*/ argc, char /*@unused@*/ *argv[]){
3     size_t a = 4711;
4     unsigned long /*@unused@*/ b = 64ul + a;
5     return 0;
6 }
```

Figure A.1: unsigned long is assigned an arbitrary unsigned integral type.

```
splint luui.c
Splint 3.1.2 --- 20 Feb 2018

luui.c: (in function main)
luui.c:4:36: Variable b initialized to type arbitrary
unsigned integral type,
expects unsigned long int: 64ul + a
To ignore type qualifiers in type comparisons use +ignorequals.

Finished checking --- 1 code warning
```

Figure A.2: Splint report on the code in A.1.

Libsodium

```
src/libsodium/include/sodium/crypto_aead_chacha20poly1305.h
    :89:61:
Function parameter k declared as manifest array (size constant
    is
meaningless) A formal parameter is declared as an array with
    size. The size of the array is ignored in this context,
    since the array formal parameter is treated as a
pointer.
```

FP: This report occurs in TweetNaCl and Libsodium. It is a matter of style, but nevertheless a good check for which a toggle is a desirable. Other similarities in the reports between Libsodium and TweetNaCl are the rotate functions "possibly using a negative right operand", and initializer blocks not explicitly specifying every array variable.

```
src/libsodium/crypto_aead/xchacha20poly1305/sodium/
    aead_xchacha20poly1305.c:38:5: Statement has no effect: (void
    )nsec
Statement has no visible effect --- no values are modified.
```

FP: Casting parameters to `void` is a common way of expressing the intention to not use an argument.

```
src/libsodium/crypto_aead/xchacha20poly1305/sodium/
    aead_xchacha20poly1305.c:39:5
2: Function crypto_stream_chacha20_ietf_ext expects arg 2 to be
    unsigned
long long gets size_t: sizeof(block0)
To allow arbitrary integral types to match long unsigned, use
+longunsignedintegral.
```

Example: The `long-unsigned-integral` flag would have been useful, as suggested by Splint.

```c
#include <stdint.h>
int main(int /*@unused@*/ argc, char /*@unused@*/ *argv[]){
    uint32_t /*@unused@*/ a = 1U;
    return 0;
}
```

Figure A.3: Assignment of an unsigned int to a fixed width 32-bit unsigned int.

```
splint fixedwidth.c
Splint 3.1.2 --- 20 Feb 2018

fixedwidth.c: (in function main)
fixedwidth.c:3:31: Variable a initialized to type unsigned int,
expects uint32_t: 1U
To allow arbitrary integral types to match any integral type,
use +matchanyintegral.

Finished checking --- 1 code warning
```

Figure A.4: Splint report on the code in A.3.

```
src/libsodium/crypto_aead/xchacha20poly1305/sodium/
    aead_xchacha20poly1305.c:46:6
2: Function crypto_stream_chacha20_ietf_ext_xor_ic expects arg 5
     to be
uint32_t gets unsigned int: 1U
```

Example: fixed width integers also require additional flags whenever they are assigned such as in this report.
The same assignment but isolated, shown in Figure A.3, generates the report in Figure A.4. A separate instance is helpful, as Splint only suggest a specific flag once.
The suggestion is to use the `match-any-integral` flag. This case is interesting, as you generally would not want to match fixed width integers to the standard ones.

TweetNaCl

```
tweetnacl.c:14:12: Initializer block for _9 has 1 element, but
    declared as u8
[32]: 9
Initializer does not define all elements of a declared array.
tweetnacl.c:14:13: Initial value of _9[0] is type int, expects
    u8: 9
Types are incompatible.
```

1. FP: One of many reports that have to do with the fact that TweetNaCl is intended to be as short as possible. Without this restriction, one could explicitly assign the other 31 bytes to be `0`. Doing so is not required, as this behavior is documented in C11, section 6.7.9, as "the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration" [18].

2. Example: As shown before, unless one uses the `charint` flag, only character assignments do not generate warnings.

```
tweetnacl.c:25:44: Right operand of << may be negative (int): x
    << c
The right operand to a shift operator may be negative (behavior
    undefined).
tweetnacl.c: (in function ts64)
tweetnacl.c:57:14: Operands of < have incompatible types (u32,
    int): i < n
```

1. FP: The rotate is only used in four lines with constants for `c`. In other types of projects (e.g. with diverse rotate usage or more contributors), a mask may be preferred.

2. FP: The report is useful for finding the type mismatch bug discussed in section 3.1. The types by themselves generally show whether something might be wrong.

```
tweetnacl.c:97:7: Test expression for if not boolean, type int:
    h
Test expression type is not boolean or int.
tweetnacl.c:123:29: String literal with 17 characters is
    assigned to u8 [16]
(no room for null terminator): "expand 32-byte k"
tweetnacl.c:133:31: Passed storage x not completely defined (*x
    is undefined):
crypto_core_salsa20_tweet (x, ...)
Storage derivable from a parameter, return value or global is
    not defined.
Use /*@out@*/ to denote passed or returned storage which need
    not be defined.
```

A couple more warnings pointing out cases that are infeasible to circumvent in TweetNaCl due to the character limit:

1. FP: The `hsalsa` flag is not checked by `h == 0`, but is instead directly used as a boolean. The flag `pred-bool-int` may be used to silence this warning.
2. FP: The ending `'\0'` is cut off (Not part of the Salsa20 constant words).
3. FP: `u8 x[64]` is not initialized in the caller, a common warning.

An important takeaway, especially from the last report, is that Splint is supposed to be continuously integrated in the development process, if used at all. A project should be partially shaped conform the way Splint reports, especially if one used the `standard` or `strict` checking mode. Splint could become usable as a non-continuous usage checker if it provided more control over the checkers. For instance, only reporting a subset of all comparisons, without matching the types for any other expressions, would be helpful. Current comparison control offers `ignoresigns` and `ignorequals` (ignores type qualifiers).

```
tweetnacl.c:304:10: Array element t[0] used before definition
An rvalue is used that may not be initialized to a value on some
    execution
tweetnacl.c:638:21: Right operand of >> may be negative (int):
s[i / 8] >> (i & 7)
tweetnacl.c:699:5: Assignment of i64 to u8: r[i] = x[i] & 255
```

1. FP: `t[0]` is initialized on line 299: `FOR(i,16) t[i]=n[i]`. The initialization is likely not detected due to the `typedef i64 gf[16]` or the `FOR` macro.
2. FP: The mask prevents the right operand from being negative.
3. FP: A mask within `u8` range is used for safe conversion.

A.1.4 Sans_t

BearSSL

```
src/symcipher/aes_x86ni_ctr.c: In function 'br_aes_x86ni_ctr_run
    ':
src/symcipher/aes_x86ni_ctr.c:186:1: warning: Loop ctr var
    suspected range mismatch. [-Wall]
186 | }
| ^
src/symcipher/aes_x86ni_ctr.c:179:18: warning: Loop ctr var
    suspected range mismatch. [-Wall]
179 |      buf[u] ^= tmp[u];
|              ~~~^~~
```

The for-loop surrounding line 179:
`for (u = 0; u < len; u ++)`. The types are `unsigned u` and `size_t len`. Line 179 is in the else-branch of the conditional `(len >= 64)`, so we may assume `len < 63`. Thus the wider integer is guaranteed to have a value in the range of the byte.

```
src/symcipher/aes_x86ni_ctrcbc.c: In function '
    br_aes_x86ni_ctrcbc_ctr':
src/symcipher/aes_x86ni_ctrcbc.c:252:1: warning: Loop ctr var
    suspected range mismatch. [-Wall]
252 | }
| ^
src/symcipher/aes_x86ni_ctrcbc.c:206:18: warning: Loop ctr var
    suspected range mismatch. [-Wall]
206 |      buf[u] ^= tmp[u];
|              ~~~^~~
```

Exactly the same scenario.

mbed TLS

```
ccm.c: In function 'ccm_auth_crypt':
ccm.c:146:36: warning: Loop ctr var suspected range mismatch. [-
    Wall]
146 |             (dst)[i] = (src)[i] ^ b[i];
                                        \
|                                       ~^~~
ccm.c:287:9: note: in expansion of macro 'CTR_CRYPT'
287 |         CTR_CRYPT( dst, src, use_len );
|         ^~~~~~~~~
```

The for-loop surrounding line 146: `for( i = 0; i < (len); i++ )`, inside the macro. The types are `unsigned char i` and `size_t tag_len`. The macro argument `use_len` is kept smaller or equal to `16` in line 278, by the following statement: `size_t use_len = len_left > 16 ? 16 : len_left`. Thus the wider integer is guaranteed to have a value in the range of the byte.

```
cipher.c: In function 'add_pkcs_padding':
cipher.c:602:15: warning: Loop ctr var suspected range mismatch.
    [-Wall]
602 |         output[data_len + i] = (unsigned char) padding_len
    ;
|         ~~~~~~~~~~~~~~~~~~~
cipher.c: In function 'add_one_and_zeros_padding':
cipher.c:643:15: warning: Loop ctr var suspected range mismatch.
    [-Wall]
643 |         output[data_len + i] = 0x00;
|         ~~~~~~~~~~~~~~~~~~~
cipher.c: In function 'add_zeros_and_len_padding':
cipher.c:681:15: warning: Loop ctr var suspected range mismatch.
    [-Wall]
681 |         output[data_len + i - 1] = 0x00;
|         ~~~~~~~~~~~~~~~~~~~~~~~~
```

1. The for-loop surrounding line 602:
   `for( i = 0; i < padding_len; i++ )`. The types are
   `unsigned char i` and `size_t padding_len`. `padding_len` is assigned
   `output_len - data_len`, with both operands being type `size_t`.
   Thus, the value will always be smaller than the block length. Unless
   2^8 byte blocks are used, the wider integer will have a value in the range
   of the byte.

2. Same scenario with `i` starting at `1`, as the first pad bit is set to 1.

3. Same scenario with `i` starting at `1`. The last byte is set to the padding
   length.

Libsodium

No warnings were generated for Libsodium.

TweetNaCl

```
./tweetnacl-usable/tweetnacl.c: In function '
    crypto_sign_ed25519_tweet':
./tweetnacl-usable/tweetnacl.c:723:26: warning: Loop ctr var
    suspected range mismatch. [-Wall]
723 |    FOR(i,n) sm[64 + i] = m[i];
|                          ~^~~
./tweetnacl-usable/tweetnacl.c:723:14: warning: Loop ctr var
    suspected range mismatch. [-Wall]
723 |    FOR(i,n) sm[64 + i] = m[i];
|              ~~~~~~~~~~
```

The `FOR(i,n)` macro is expanded to `for (i = 0;i < n;++i)`. The types are `i64 i` and `u64 n`. The message size would have to be 2^{63} or longer, which is unadvised and 'unlikely' to ever occur.

```
./tweetnacl-usable/tweetnacl.c: In function '
    crypto_sign_ed25519_tweet_open':
./tweetnacl-usable/tweetnacl.c:790:21: warning: Loop ctr var
    suspected range mismatch. [-Wall]
790 |    FOR(i,n) m[i] = sm[i];
|                     ~~~~~
./tweetnacl-usable/tweetnacl.c:790:13: warning: Loop ctr var
    suspected range mismatch. [-Wall]
790 |    FOR(i,n) m[i] = sm[i];
|              ~^~~
./tweetnacl-usable/tweetnacl.c:802:15: warning: Loop ctr var
    suspected range mismatch. [-Wall]
802 |      FOR(i,n) m[i] = 0;
|               ~^~~
./tweetnacl-usable/tweetnacl.c:806:21: warning: Loop ctr var
    suspected range mismatch. [-Wall]
806 |    FOR(i,n) m[i] = sm[i + 64];
|                     ~~~~~~~~~~
./tweetnacl-usable/tweetnacl.c:806:13: warning: Loop ctr var
    suspected range mismatch. [-Wall]
806 |    FOR(i,n) m[i] = sm[i + 64];
|              ~^~~
```

All three lines come from the same function, using the same loop counter variable. `n` is a parameter, which should hold a value corresponding to the size of the signed message parameter, `const u8 *sm`. The same `FOR` loop macro as before is used, this time with the types `int i` and `u64 n`. A message length of 2^{31} bytes is unadvised but reachable.