CtChecker: A Precise, Sound and Efficient Static Analysis for Constant-Time Programming

Quan Zhou ✉
Penn State University, University Park, PA, USA

Sixuan Dang ✉
Duke University, Durham, NC, USA

Danfeng Zhang ✉
Duke University, Durham, NC, USA

―――― **Abstract** ――――

Timing channel attacks are emerging as real-world threats to computer security. In cryptographic systems, an effective countermeasure against timing attacks is the constant-time programming discipline. However, strictly enforcing the discipline manually is both time-consuming and error-prone. While various tools exist for analyzing/verifying constant-time programs, they sacrifice at least one feature among precision, soundness and efficiency.

In this paper, we build CtChecker, a sound static analysis for constant-time programming. Under the hood, CtChecker uses a static information flow analysis to identify violations of constant-time discipline. Despite the common wisdom that sound, static information flow analysis lacks precision for real-world applications, we show that by enabling field-sensitivity, context-sensitivity and partial flow-sensitivity, CtChecker reports fewer false positives compared with existing sound tools. Evaluation on real-world cryptographic systems shows that CtChecker analyzes 24K lines of source code in under one minute. Moreover, CtChecker reveals that some repaired code generated by program rewriters supposedly remove timing channels are still not constant-time.

1 Introduction

Modern cryptographic systems are vulnerable to timing attacks [21, 10, 26, 6], which can quickly reveal confidential keys by analyzing the encryption/decryption time of those systems. While the treat has been well-known for decades, identifying timing channel vulnerabilities in cryptographic systems is a daunting task, as timing channels result from implementation details such as data and instruction cache effects, branch prediction buffers and memory controllers. As all of those hardware features are invisible in the source code, manually identifying timing channel vulnerabilities is extremely challenging, if possible at all, as doing so precisely requires a crystal clear view of the whole software-hardware stack and how secret information flows throughout the stack.

To make it feasible to mitigate timing channels, one common practice is to identify and rule out *dangerous code patterns* that lead to timing channels. Notably in cryptographic systems, a common countermeasure against timing attacks is to follow *constant-time disciplines* [1, 11], which rules out (1) branching on secret-dependent data, as well as (2) accessing memory with secret-dependent offset (e.g., an array access with a secret-dependent index). For example, a secret-dependent branch `if s[i] then x = 2` may be identified by noting that `s[i]` is the *i*-th bit of the private key, hence violating the first constant-time discipline. We might further replaced it with `x = s[i]*(2-x)+x`, or `x = (-s[i]&2)|((s[i]-1)&x)`, which are both functionally equivalent to the original code. While the violation in this example is easy to spot as it directly uses the secret value *s*, detecting constant-time violations and rewriting them in a secure way in general is still error-prone, as evidenced by timing leak in manually validated code [27] as well as a sequence of timing-related patches where early patches introduce new vulnerabilities that are fixed by later ones [36]. Hence, more rigorous and automated techniques are necessary for security.

Motivated by the need for rigorous and automated tools, designing and developing automated tools for detecting and sometimes repairing constant-time violations has been an active research area. Observing the relation between constant-time disciplines and non-interference [16], most automated tools rely on some form of information flow analysis to detect constant-time violations (i.e., to detect "tainted" branch conditions and memory addresses). For example, ct-verif [1] uses a sound and complete reduction on the source code to verify constant-time disciplines as a safety problem, via off-the-shelf verification tools. FACT [11] is a domain-specific language that uses a static information flow type system to detect constant-time violations. Constantine [7] deploys a dynamic taint analysis [7], while FlowTracker [32] uses an optimized program dependence graph (PDG) to identify constant-time violations. If we narrow down the scope further and focus on *cache-based* timing channels only, it is also common to identify constant-time violations first, before further refining the results to those that are vulnerable to cache-based attacks only. For instance, SC-Eliminator [46] uses a static taint analysis to identify "leaky conditional statements" (i.e., sensitive branches) and "leaky lookup-table accesses" (i.e., sensitive memory addresses via array accesses) before applying a (more costly) static cache analysis on leaky lookup-table accesses to filter out the ones that are not vulnerable to cache-based timing attacks (though they might still be vulnerable to other variants of timing attacks; see Section 2.3). The same applies to SpecSafe [9] and oo7 [43], which used static and dynamic taint analysis respectively before applying more costly cache analysis to refine their results.

With ample tools that use information flow analysis to detect constant-time violations, the *precision issue of those tools* is still under-investigated to the best of our knowledge. Yet, to be practical, the precision issue is just as important as soundness, especially given the common wisdom that static information flow analysis usually comes with high false positive rates when being applied to real-world applications [33]. To be more concrete, the following research questions still linger when cryptographic library developers plan to develop or adopt an information flow analysis for the purpose of detecting constant-time violations:

RQ1: To analyze cryptographic libraries, what are the impacts of various design decisions (e.g., field-sensitivity, declassification, context-sensitivity) on the analysis precision?

RQ2: Which approach (e.g., logic-based formal verification, PDG-based analysis) should they follow in order to achieve better trade-offs between precision and performance?

RQ3: Is it possible to improve the precision of existing tools without sacrificing soundness and performance?

To address these research questions, we start from a baseline PDG-based static information flow analysis called PIDGIN [18], with minor extensions that use its resulting sensitivity of registers and memory blocks to detect violations of constant-time disciplines. With the baseline implementation, we thoroughly study the root causes of false positives produced by the baseline and improve it with various features such as field-sensitivity, declassification and flow-sensitivity to reduce its false positive rate. The precision study is performed on a benchmark consisting of various implementations of modular exponentiation used in popular cryptographic systems (Libgcrypt, OpenSSL, mbedTLS and BearSSL), which are known to be vulnerable to timing attacks, as well as a set of automatically repaired code from Constantine [7], which are expected to be constant-time. The precision study shows that field-sensitivity and declassification are the most effective features. As a result from the precision study, we built CtChecker, a precise, sound and efficient information flow analysis for constant-time programming. CtChecker was implemented based on PIDGIN, with several precision improving features introduced. CtChecker targets LLVM intermediate representation (IR), which allows it to analyze various source code languages with compatible compiler front-ends. In summary, CtChecker reduces the false positives of the baseline by 67.6%, and we observed that the remaining ones are mainly due to imprecision in the sound points-to analysis used by CtChecker. Moreover, CtChecker detects true positives in 6 repaired programs produced by Constantine, which was not revealed before to the best of our knowledge.

A1: Field-sensitivity and declassification are the most effective features that can improve analysis precision, while the precision of point-to analysis (used by sound information flow analysis) has a considerable impact on the overall precision.

To study the remaining two research questions, we first compare the precision of CtChecker with respect to ct-verif [1], which first transforms source code to target code via a sound and complete reduction, and then verifies safety properties on the target code via off-the-shelf verification tools. Despite the fact that the reduction is sound and complete, and the common wisdom that logic-based formal verification generally offers superior precision than static program analyses that are built on approximation of program semantics, CtChecker turns out to be both more precise and more efficient compared with ct-verif [1]. Digging into the results, we found that ct-verif uses *loop invariant heuristics* to verify the code after reduction, which introduced higher false positive rates than CtChecker. Moreover, PDG-based approach is more appealing for detecting constant-time violations as it can pinpoint those violations in the code, while verifiers only produce a binary result of the existence or absence of violations.

A2: Based on our empirical study on state-of-the-art tools, we found that PDG-based information flow analysis offers the best trade-off between precision and performance compared with logic-based approach.

We further compare CtChecker with CacheS [44] and SC-Eliminator [46] on the benchmarks that those tools were evaluated on. Both tools are *soundy* as CacheS is built on lightweight but unsound memory model, and SC-Eliminator assumes no information flow via aliasing. Though CtChecker is built on a sound points-to analysis, we found that CtChecker reports very similar positives with those reported by CacheS [44], even though CacheS is built on complex abstraction interpretation with an unsound memory model. Compared with the static taint analysis used by SC-Eliminator, CtChecker reports fewer positives in 6 programs and the same positives in 9 programs in the SC-Eliminator benchmark. CtChecker reports more positives in 2 programs, but they are all true positives missed by SC-Eliminator, since it does not propagate information flows via aliasing.

A3: CtChecker improves the precision of existing tools without sacrificing soundness and performance. Compared with CtChecker, existing solutions either fall short on the high false positive rates with sound static method [1], or inherit the unsoundness of dynamic

method [45, 37], or simplified but unsound memory model [44, 46]. Notably, the precision of CtChecker is close to those built on unsound memory models, even though they have the advantage of possibly reporting fewer false positives by sacrificing soundness.

In summary, this paper makes the following contributions:

- Design and implementation of CtChecker, a precise, sound and efficient static information flow analysis for constant-time programming. The source code is made publicly available[1].
- Identification of the imprecision sources (e.g., field-sensitivity, declassification and flow-sensitivity) of constant-time analysis on cryptographic libraries (Section 3), and improvement of overall analysis precision based on the findings. Overall, field-sensitivity and declassification are the most effective features that improve precision, while combining multiple features enables CtChecker to reduce the total false positives compared to the baseline by 67.6% (Section 4).
- Comparison between CtChecker and state-of-the-art tools with similar goals. Evaluation results suggest that PDG-based information flow analysis offers the best trade-off between precision and performance compared with logic-based approach (e.g., ct-verif [1]) and abstract interpretation (e.g., CacheS [44]). Moreover, its precision is close to tools that are built on unsound memory models (Section 4).
- Evaluation of CtChecker on automatically repaired code from Constantine [7] reveals new timing channels that are not reported before (Section 4).

2   Background

2.1   Information Flow Analysis

Information flow analysis tracks interactions of information throughout a program. Given the confidentiality of program inputs, an information flow analysis tracks which data have been computed from confidential information or its derivatives. In general, information flow analysis handles complex confidentiality and/or integrity policies which can be formalized as a security lattice [13], while its simplest setting, known as *taint analysis*, only handles a two-level confidentiality lattice with "public" and "secret" labels only. A *sound* information flow analysis typically enforces *non-interference* [16] or its variants. Intuitively, non-interference guarantees that information labeled with higher confidentiality (or lower integrity) has no influence on information labeled with lower confidentiality (or higher integrity).

Listing 1 Example of Explicit and Implicit Information Flow.

```
// key = sensitive information
x = key + 1;
y = 0;
if (key > 100) {
  y = 1;
}
```

There are two kinds of information flows: explicit and implicit, as illustrated in the example shown in Listing 1. In this simple example, key is the only confidential input. We can see that x is directly computed from key, forming an *explicit* information flow. On the other hand, the variable y is assigned to a public constant in the true branch. However, due to the fact that the assignment occurs only in a branch whose condition is dependent on key, an attacker with the ability to observe the value of y can infer if the value of key is above 100 or not. In this case, the confidential data *implicitly* flows into y.

---

[1] `https://github.com/psuplus/CtChecker`

A variety of information flow analyses have been implemented with different methodical approaches. The utilization of program dependence graph (PDG) to detect the flow of information inside a program can be found in works such as PIDGIN [18] and FlowTracker [32]. Another approach is to use reduction techniques such as self-composition [5, 1] and product programs [4, 48] to reduce the information flow problem to safety properties, which can be verified by formal methods. Type-based approach, another more traditional method, can be found in various implementations [23, 42, 30]. Besides the static sound approaches above, dynamic taint analysis tracks information flow [14, 24, 39] at program execution time.

In this paper, we focus on the PDG-based approach and logic-based approach as they have been used in existing *automated and sound* tools that detects timing channels [1, 32]. Some prior work uses type-based approach to detect timing channels in software [49] and hardware [51] designs, but the precision of type systems is usually limited [33, 20]. They require type annotations from programmers, which is time-consuming, and pinpointing the root cause of type errors is a nontrivial task [50, 25, 34].

2.2 Timing Channels in Cryptosystems

A timing channel is a side channel in which an attacker uses program execution time to learn information about sensitive data. In some implementations of sliding window exponentiation, for example, the sequences of squares and multiplies can be measured due to differences in each method's execution time. This attack can be illustrated in the source code from an old OpenSSL implementation of sliding window exponentiation, shown in Listing 2.

Listing 2 Square and Multiply Timing Channel.

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

The for-loop here iterates through each bit in the confidential exponent `p`. In each iteration, it first computes the square of `v`, and if the i-th bit of `p` is set, an additional multiplication computation is executed before proceeding to the next loop iteration. Since the extra multiplication computation is *only performed* when the i-th bit of `p` is set, the code is vulnerable to a timing attack which utilizes the "side effects" of the extra computation on timing to rebuild the entire private key (e.g., [6]).

Another common kind of timing channels in cryptography algorithm implementations root from array accesses being indexed with an offset derived from secret. The reason is that when accessing the memory, different indices may cause the loading or eviction of data into or from different cache lines in the data cache. By observing such behaviors, an attacker could reveal secret data; this variant of timing attacks is also known as *cache attacks*. The code snippet below is an example which is vulnerable to cache attack, where a public array `Sbox` is accessed with secret key `RK`:

```
RK[4] = RK[0] ^ Sbox[(RK[3] >> 8) & 0xFF];
```

As different values of `RK[3]` can lead to different cache lines to be fetched, a cache-probing attack can be launched to learn the value of `RK[3]`. The code snippet represents real timing vulnerabilities in encryption algorithms such as AES [17].

2.3   Constant-Time Disciplines and Cache-Specific Analysis

As timing channels are revealed by the execution time of a program, and many hardware features (e.g., data/instruction cache, CPU pipeline and cache bank) can affect timing, manually reasoning about timing channel vulnerabilities is extremely challenging, if possible at all. So as a practical countermeasure against timing channels, the threat model of constant-time discipline [1, 11] defines two constant-time violations of programs which can be exploited by attackers: (1) secret-dependent branch conditions, and (2) secret-dependent memory addresses. The constant-time disciplines are widely adopted in security-critical cryptographic systems [1, 11] due to a few benefits:

- It is *agnostic* to hardware configurations (e.g., cache configuration and replacement policy) and features (e.g., cache, pipeline and cache bank) that are utilized by timing attacks.

- It provides a security abstraction that is more attractable both for programmers and for rigorous program analysis. For example, it is a common practice to use information flow analysis [1, 7, 11, 32] to detect constant-time discipline violations: it is sufficient to tag branches and memory accesses that use tainted values.

We note that under a *weaker threat model* that focuses on cache-based timing attacks only, various cache analyses [8, 35, 46] has been developed to detect if confidential information has impact on the cache hit/miss behaviors. For instance, SC-Eliminator [46] assumes a weaker threat model where an attacker only observes the number/type of instructions and cache hits/misses. Consider `preload A; A[secret]`, where `preload A` loads the whole array `A` into the cache. The code is secure per SC-Eliminator's threat model since `A[secret]` always hits the cache. But it is insecure against other sophisticated attacks, such as the CacheBleed attack [47] that exploits cache-bank conflicts, and traffic analysis on the memory bus.

Despite the differences in their threat models, we note that cache analysis sometimes performs a static taint analysis first to identify array indices that are potentially tainted by sensitive data. Then, with the results from the taint analysis, a more costly cache analysis is performed to determine cache hits/misses. A positive from the taint analysis is removed if the cache analysis decides that this positive is a cache must-hit [35, 46]. Hence, precision improvements in sound and efficient static taint analysis, the focus of this paper, could also improve the precision of existing cache analysis, despite their very different threat models.

3   CtChecker Design

In this section, we depict the design details of CtChecker. We first describe the general workflow of CtChecker. Then we highlight the unique challenges and their solutions, and finally discuss the precision of CtChecker.

3.1   General Workflow

CtChecker first captures information flows by identifying the information flow introduced by each instruction according to its semantics. The captured flows are represented as a set of constraints, where each constraint element represents the sensitivity of registers or memory blocks. A least solution that satisfies all constraints is then computed. Lastly, all branch conditions and memory accesses are checked against the least solution to see if any sensitive information was used in them, resulting in violations of constant-time discipline.

```
<pointer> = getelementptr inbounds <ty>, ptr
    <ptrval>{, [inrange] <ty> <idx>}*
```

Flow: $V(\text{ptrval}) \to V(\text{pointer})$

```
store <ty> <value>, ptr <pointer>
```

Flow: $V(\text{value}) \to D(\text{pointer})$

```
<result> = load <ty>, ptr <pointer>
```

Flow: $V(\text{pointer}) \cup D(\text{pointer}) \to V(\text{result})$

Figure 1 Memory-Related Specification for LLVM IR.

3.1.1 Capturing Information Flows

The information flow specification of each instruction describes the sources (i.e., where information come from) and the sinks (i.e., where information go to). Based on the semantics of each instruction, the sources and sinks are usually easy to identify: instruction operands being read are sources and operands that are written to are sinks. However, since pointers can point to different data (in the memory) and the data to which they point to can change, a concrete information flow specification needs to distinguish three categories of elements, namely 1) the operand itself (e.g., a register), 2) the memory location that a pointer directly points to, and 3) all memory locations that are reachable from a pointer (through pointer arithmetics). Hence, we define three functions (V, D, R) which return the elements to be constrained for a value, respective to the three categories.

- $V(\text{x})$, the value associated with the operand x.
- $D(\text{p})$, the memory block that a pointer p points to.
- $R(\text{p})$, the set of all reachable (i.e., accessible) memory blocks from a pointer p.

In order to correctly compute D and R, a sound points-to analysis is employed. Formally, the points-to analysis creates a directed graph $G = (P, M, E)$ from the code being analyzed, where node set P represents all pointer values, node set M represents all memory blocks being allocated, and edge set E links nodes in P to nodes in M (i.e., the points-to relation), as well as links connecting memory blocks, as a memory block can also hold a pointer.

To compute the function $D(\text{p})$, we locate $\text{p} \in P$ and return the set $\{m \mid m \in M \land (\text{p}, m) \in E\}$. Similarly, to compute the function $R(\text{p})$, we return the set of memory blocks that are *reachable* from p in G. For example, assume a pointer p_1 pointing to a pointer field of a data structure, $D(\text{p}_1)$ will be the memory block storing the pointer only, while $R(\text{p}_1)$ contains all the memory blocks that can be recursively reached by p_1 through node traversal, which include both the memory block storing the pointer and the data that the pointer points to.

For most IR instructions, information flow is captured by a simple value flow, $V(\text{source}) \to V(\text{sink})$. For memory-related instructions (e.g., the `getelementptr` (`GEP`), `store` and `load` instructions in LLVM IR), the more complicated specification is given in Figure 1. Specifically, for `GEP` instruction that calculates the memory address of a subelement in an aggregate data structure (i.e., array and structure) from the base pointer and the index of the target subelement, the sink $V(\text{pointer})$ is the address directly pointed to by p. For `store` and `load`, function D is used to specify the source and destination of the memory write and read respectively. Function R is used for function calls, which we discuss in Section 3.2.4.

3.1.2   Constraints and Their Least Solution

With the information flow specification, CtChecker creates a set of *constraints* from sources to
sinks for each instruction. For each register and memory block $m \in M$, a distinguished *con-
straint element* is created (i.e., a one-to-one mapping). For an information flow $source \to sink$,
a constraint is generated as $E_{source} \le E_{sink}$ where E_{source} and E_{sink} are the corresponding
constraint element of the *source* and *sink* respectively. The $\le$ symbol represents a partial
ordering on the constraint elements and each constraint element can either be L (public) or
H (secret), where $L < H$ and $H \nleq L$. Additionally, CtChecker identifies the initial sensitive
data m (e.g., secret keys) and generates constraints of the form $H \le E_m$. For example,
code snippet "k=secret; x=k; y=0;" will generate three constraints $H \le E_k$, $E_k \le E_x$
and $0(\text{constant}) \le E_y$. These generated constraints are then added to the set of constraints,
which are used to find the least solution.

The least solution, if exists, can be computed with linear-time algorithms, such as
the Rehof-Mogensen algorithm [31]. By definition, the *least solution* provides the *least*
confidentiality level of each constraint element, where the set of constraint elements with the
least level H are considered to contain sensitive information. For the code snippet above,
the least confidentiality level of k, x and y are H, H, and L, respectively. As a result, k and
x are considered containing sensitive information.

3.1.3   Checking the Constant-Time Discipline

With the least solution at hand, CtChecker can check whether the analyzed program adheres
to the constant-time discipline by examining all branch conditions and memory accesses,
where a violation is found if a branch condition has level H, or a memory address being
loaded from or stored to has level H. CtChecker reports all locations in terms of the line
number in the source code regarding violations of the constant-time discipline.

3.2   Challenges and Solutions

The general workflow above presents the foundation of a *sound* analysis of constant-time
programs. However, to develop an *useful* analysis, we need to address the following challenges.

3.2.1   Field-Sensitivity

In the naive analysis above, function D is modeled as $D(\text{pointer})$, the aggregated data
structure that pointer points to. However, it is common for cryptographic libraries to store
secret and public information in the same structure. In Listing 3, consider the `gcry_mpi`
struct of Libgcrypt, where the array under `expo->d` in the modular exponentiation function
holds secret value, while all other metadata fields are public. The naive approach would
constrain `expo` as a single entity, reporting both branches at lines 8 and 9 as secret-dependent
branches. However, the branch on `expo->flags` is not secret, resulting in a false positive.

To address the issue, both the sound points-to analysis and information flow analysis
need to be *field-sensitive*, meaning that they both need to differentiate different fields in
a structure. In particular, points-to analysis creates one separate memory block for each
subelement in aggregate data structures, and correspondingly, CtChecker creates one distinct
constraint element for each memory block from points-to analysis. By retrieving the offset
and size information when possible (using `idx` operand from the corresponding `GEP` instruction
of a `store` or `load` instruction as well as type information), CtChecker refines function D as
$D(\text{pointer}, \text{offset}, \text{size})$, providing the necessary lookup information into the memory nodes in
G, constructed by a field-sensitive points-to analysis. Function R is refined in a similar way.

■ **Listing 3** Field-Sensitivity Issues.

```
1  struct gcry_mpi {
2    ...
3    unsigned int flags;
4    mpi_limb_t *d;        // secret value
5  };
6
7  struct gcry_mpi *expo;
8  if (expo->flags) ... // not secret-dependent
9  if (expo->d[i])   ... // secret-dependent
10 if (expo->d)      ... // not secret-dependent
```

One remaining subtlety is to distinguish pointer values and the memory blocks that they point to. For example, it is common to store private keys at the public memory address (i.e., revealing the addresses of the keys does not reveal their values). To prevent CtChecker from over-tainting these addresses (e.g., line 10 in Listing 3), CtChecker follows a two-phase approach. In the first phase, CtChecker propagates the addresses that are potentially storing sensitive information. In the second phase, when the data in these addresses is accessed, the tainted information will then be tracked from these accesses.

We note that while field-sensitivity can be enabled on many cases, a sound analysis inevitably needs to sacrifice field-sensitivity in cases where type information is missing, or aliases have inconsistent types, for instance. In these cases, refined function $D(\mathsf{pointer}, \mathsf{offset}, \mathsf{size})$ resorts to its basic version without offset and size (i.e., D retrieves all fields in the structure that $\mathsf{pointer}$ points to).

3.2.2 Declassification

In real-world applications, strict information flow analysis should be relaxed to allow information flows that reveal limited or intended amount of sensitive information. This relaxation is known as *declassification*. CtChecker supports a *whitelisting* mechanism, where a user-provided whitelist (often provided by a programmer with domain knowledge) consists of variables that are derived from tainted data but are considered harmless. For all cryptographic libraries, we add variables storing *key sizes* but nothing else to the whitelist. These variables are manually checked to make sure that they do not contain key content themselves.

With a whitelist, CtChecker removes constraints that are associated with whitelisted variables after constraint generation. This way, not only the whitelisted variables are considered public, but also variables derived from them. The branches based on whitelisted variables or their derivatives are not reported by CtChecker.

3.2.3 Flow-Sensitivity

A variable might store both sensitive and public values at different program points, leading to imprecision issues. Consider the code snippet where both branches are dependent on `i`.

```
1  int i = key;
2  if (i == 0) ... // secret-dependent
3  i = 10;
4  if (i == 0) ... // not secret-dependent
```

A naive information flow analysis generates the constraints $H \leq E_i$ and $L \leq E_i$, where `i` throughout the program shares the same constraint element E_i. The least solution of the constraints is $E_i = H$, meaning that `i` is potentially sensitive. As a consequence, both branch conditions at lines 2 and 4 are marked as violations of constant-time discipline.

```
declare <RetType> @<FnName> ([arg list])
```

Flow: $V_{args} \cup R_{args} \rightarrow T_{ret} \cup R_{args}$

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

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

$$T_{ret} = \begin{cases} V(\mathsf{retval}) & \mathsf{RetType} = \mathsf{primitive} \\ R(\mathsf{retptr}) & \mathsf{RetType} = \mathsf{pointer} \end{cases}$$

Figure 2 Unknown Function Specification for LLVM IR.

Observing that the root cause of the issue above is lacking flow-sensitivity, CtChecker utilizes existing compiler support to improve flow-sensitivity. In LLVM, all registers are in static single-assignment (SSA) form. Hence, CtChecker can utilize LLVM's `mem2reg` pass, which transforms the IR code by turning the standard `alloca-store-load` instruction sequences on memory (e.g., assignment and use of `i` in the example above) into simple register assignments. With the code transformation, the two `i`'s are named as `i.0` and `i.1`, respectively. Hence, two constraints are generated: $H \leq E_{i.0}$, $L \leq E_{i.1}$, and the least solution is $E_{i.0} = H, E_{i.1} = L$. Therefore, only the branch at line 2 which uses `i.0` is marked.

However, we note that due to aliasing and other subtleties in C language's memory model, enabling flow-sensitivity on pointer-based accesses while maintaining soundness is much more challenging, which is beyond the scope of this paper.

3.2.4   Unknown Functions

The analyzed code often calls to external functions whose source code is either unavailable, or not covered by the analysis. To soundly capture information flows with the absence of some function definitions, we need to constrain possible flows in those missing functions. Obviously, input arguments can flow to return values. Moreover, if an argument or the return value is a pointer, any value that is *reachable* from the pointer-argument might flow to all *reachable* values from the pointer-return (e.g., through pointer arithmetics and memory writes). Hence, the information flows with absent function implementation is specified as the rule in Figure 2, where reachable memory from pointer-arguments are both sources and sinks of information flow, while reachable memory from pointer-return are sinks.

3.3   Precision of CtChecker

While CtChecker is empowered by various techniques above to improve its precision while maintaining soundness, false positives are still unavoidable like any nontrivial static program analysis.

One potential source is from a sound points-to analysis. To be sound, the points-to analysis must mark memory nodes as collapsed when the type information is inconsistent or missing. Once a node is collapsed in the points-to analysis, field-sensitivity is lost on the memory node. Moreover, whenever the points-to analysis fails to distinguish two different pointers' corresponding memory nodes, it merges them into one node, resulting in an over-approximation of the taint analysis.

Second, `CtChecker` is a context-sensitive interprocedural analysis. However, when a callee function is invoked multiple times within the same caller function with different arguments, `CtChecker` only creates one context for all calls. This leads to the same indexed arguments of the multiple calls being aliased in the points-to analysis. As a result, a similar over-approximation is observed.

Furthermore, the adoption of LLVM's `mem2reg` pass only enables flow-sensitivity on non-aggregate type memory. Memory accesses involving GEPs are still flow-insensitive.

While the above limitation prevents us from removing all false positives, we observe that `CtChecker` is able to outperform existing sound analysis. We provide the evaluation details next in Section 4.

4   Evaluation

4.1   Implementation

We implement `CtChecker` on PIDGIN [18], a static information flow analysis that integrates a query language into program dependence graphs (PDGs). However, PIDGIN lacks the precision-enhancing features discussed in Section 3.2, which we implement with an additional 2100 LOC in C++.

One implementation choice of static information flow analysis is the points-to analysis to derive sound approximation of memory blocks that a pointer might point to. The two mainstream points-to analysis algorithms are the unification-based Steensgaard's algorithm [38] and the inclusion-based Andersen's algorithm [2]. Both algorithms have implementations that are both context- and field-sensitive in order to gain better precision. For example, DSA [22] is a field- and context-sensitive points-to analysis based on Steensgaard's algorithm with heap cloning. On the other hand, Andersen's algorithm is generally considered more precise but also costly. Tools such as SVF [40] provide precision from context- and field-sensitivity without sacrificing much performance. Noting that DSA is used in our baseline PIDGIN and ct-verif [1], a representative logic-based tool, `CtChecker` is also built on top of DSA to make fair comparison with them (see the comparison with PIDGIN in Section 4.4 and ct-verif in Section 4.5). We leave the study of the impact of points-to analysis on constant-time analysis as future work.

4.2   Benchmarks

We evaluated `CtChecker` on two sets of benchmark programs. The first benchmark set consists of code taken from four cryptographic libraries, i.e., BearSSL v0.6 [29], Libgcrypt v1.10.1 [12], mbedTLS v3.2.1 [41] and OpenSSL v1.1.1q [15]. Among the four libraries, Libgcrypt and OpenSSL have widespread use, mbedTLS is built for embedded platforms, and BearSSL is less popular but it claims to be a constant-time cryptographic library [28]. In particular, the code consists of the modular exponentiation implementations of each library, where 4 implementations are taken from OpenSSL, as it is the only library that has various implementations for the same functionality. In the benchmark, the exponents in the modular exponentiation computation is marked as confidential sources. According to the definition of constant-time discipline, the sinks of the analysis are simply branch conditions and memory addresses.

The second benchmark set consists of code generated by a constant-time rewriter Constantine [7], which automatically identifies timing channels in the source code and repair them to follow the constant-time discipline[2]. To the best of our knowledge, this is the first work to analyze the rewritten code by constant-time rewriters. Due to an incompatible LLVM version used by Constantine [7], we used an off-the-shelf C-backend [19] to translate their rewritten IR back to C source code whenever possible.

4.3   Evaluation Setup

We answer four research questions with the evaluation:

Q1: Impact of separate features. How do field-sensitivity, declassification and flow-sensitivity affect the analysis precision as separate features? Will including additional source code improve the analysis precision? Does CtChecker improve the precision of its baseline (i.e., PIDGIN)?

Q2: Comparison with state-of-the-art. Does CtChecker improve the precision when compared with ct-verif [1], a sound verifier for constant-time programming? Does CtChecker produce comparable or even more precise results when compared with CacheS [44] and SC-Eliminator [46]), both are built on simplified but unsound memory model?

Q3: Precision. How many false positives does CtChecker produce? What are the origins of the remaining false positives?

Q4: Scalability. Does CtChecker scale to real-world codebase with moderate size?

4.4   Impacts of Analysis Features

To answer Q1, we first create three variants of CtChecker, where only one feature among field-sensitivity (FS), white-list (WL) and flow-sensitivity (FL) is enabled. One extra feature, which is external to CtChecker, is how much code does it cover in the analysis. To study the impact of code coverage, we create two versions of each library implementation: one only includes the essential code that is necessary to compile the modular exponentiation implementation, while the other version (SRC) includes utility functions such as the multi-precision integer (mpi) or big number (bn) libraries[3]. The evaluation results are summarized in Table 1.

The improvement for field-sensitivity alone (column FS) is smaller than expected: while all implementations allocate secret and non-secret values in same structures, only four implementations (Libgcrypt, mbedTLS, BearSSL and OpenSSL-MontConstTime) observe some improvements. The reason is largely due to the lack of utility function implementations: both points-to analysis and information flow analysis remain very conservative without callee's implementation, making it hard to differentiate read/write effects on each individual data field.

All four libraries saw a considerable reduction in positives when whitelist was used (column WL). The removed positives only leak key sizes, which are explicitly declassified in the whitelist. OpenSSL and mbedTLS saw a slight reduction with flow-sensitivity enabled (column FL).

---

[2] We pick Constantine [7] instead of other available rewriters such as SC-Eliminator [46], Lif [35] as they both assume a weaker threat model that only tackles cache attacks (see Section 2.3). In particular, their rewritten code with prefetching technique still violates constant-time disciplines.

[3] The only exception is on BearSSL, which remains the same for both versions for two reasons: (1) the modular exponentiation function only contains high-level code that makes up fewer than 20 lines of code. An analysis on it alone does not generate any meaningful result, and (2) BearSSL has a much smaller codebase compared with other libraries.

■ **Table 1** Number of positives based on features. Each column represents the analysis result with some features enabled. *Base*: the baseline analysis, *FS*: field-sensitivity, *WL*: whitelist, *SRC*: adding extra source code, *FL*: flow-sensitivity, and *All*: all features enabled. *TP* represents the true positives and *Reduction* computes the reduction rate of false positives, i.e., (Base − All)/(Base − TP).

Library	Base	FS	WL	SRC	FL	All	TP	Reduction
Libgcrypt 1.10.1	66	46	55	76	66	30	6	60.0%
mbedTLS 3.2.1	50	45	48	33	45	10	4	87.0%
BearSSL 0.6	18	15	6	18	16	3	1	88.2%
OpenSSL 1.1.1q								
Reciprocal	14	14	3	20	13	9	2	41.7%
Mont	45	45	36	37	44	25	2	46.5%
MontConstTime	36	27	28	29	34	18	0	50.0%
MontWord	4	4	2	15	4	12	1	-267%
binsec/aes_big	0	0	–	–	0	0	0	–
binsec/des_tab	51	26	–	–	51	26	24	92.6%
binsec/tls-rempad-luk13	7	6	–	–	7	6	6	100%
appliedCryp/3way	41	0	–	–	41	0	0	100%
appliedCryp/des	72	62	–	–	72	62	62	100%
appliedCryp/loki91	75	72	–	–	75	72	56	15.8%
ghostrider/findmax	0	0	–	–	0	0	0	–
ghostrider/matmul	4	0	–	–	4	0	0	100%
libg/des	448	432	–	–	448	432	432	100%
pycrypto/ARC4	20	19	–	–	20	19	19	100%
Overall	951	813	178	228	940	724	615	67.6%

Including additional source code (column SRC) does not necessarily reduce false positives: doing so in fact increases positive numbers for Libgcrypt, OpenSSL-Reciprocal and OpenSSL-MontWord. This somewhat surprising degradation comes from the imprecision of the underlying points-to analysis. The points-to analysis, using heap cloning technique, will merge distinct nodes that are processed by a common function. The analysis also merges nodes that are found to be in the same equivalence class. In the case of Libgcrypt, nodes that were considered distinct in the baseline test, end up aliased to the same node in the full source benchmark.

CtChecker enables all features and analyzes more than the essential code (i.e., it also covers utility function implementations), whose result is shown under the column "All". For most libraries, CtChecker delivers the most precise result, with the exceptions of OpenSSL-Reciprocal and OpenSSL-MontWord. By inspecting the differences, we concluded that the reason is also due to undesirable effects in the points-to analysis when additional code is being analyzed.

4.5 Comparison with ct-verif

ct-verif [1] is one of the first sound tools for verifying constant-time properties; it uses the self-composition technique [5] to convert the constant-time property into a classical program verification problem. One reason for comparing with ct-verif is that it is also built on top of the DSA [22] points-to analysis; hence, the comparison focuses on the advantages of each approach, rather than some engineering details, such as the difference in the points-to analysis, in their implementations.

■ **Table 2** Comparison with ct-verif. "Full-SRC" corresponds to full source in Table 1. "No-Undefined-Function" is the version where all function calls without sources are removed. "ct-verif-Verified" stands for all positives in ct-verif are removed, while "CtChecker-Verified" stands for all positives in CtChecker are removed.

	ct-verif				CtChecker			
Library	Full SRC	No Undefined Function	ct-verif Verified	CtChecker Verified	Full SRC	No Undefined Function	ct-verif Verified	CtChecker Verified
Libgcrypt 1.10.1	–	20	0	10	30	6	0	0
mbedTLS 3.2.1	OOM	5	0	1	10	4	0	0
BearSSL 0.6	3	3	0	0	3	3	0	0
OpenSSL 1.1.1q								
Reciprocal	–	2	0	0	9	2	0	0
Mont.	–	4	0	2	25	2	0	0
Mont. Const. Time	–	2	0	3	18	0	0	0
Mont. Word	–	1	0	0	12	1	0	0

One challenge in the comparison is that as a verification tool, ct-verif only reports whether the input program is constant-time or not[4]. To find the exact lines that ct-verif deems constant-time violations, a line-by-line check on the source code is needed. Whenever ct-verif reports a positive, we log the current line, modify it with some constant-time code, and run ct-verif again. The same strategy is applied to function calls that lead the control flow to other functions.

Even though we carefully make the changes so that the information flow remains the same, there is still a chance that the information flow might be changed while rewriting the code that has constant-time violations. For a fair comparison, both tools are running on the same rewritten code.

4.5.1 Results

Both ct-verif and CtChecker are sound analyses and we did observe that both tools report all true positives. The differences are on false positives. To evaluate the false positive reported by each tool, we consider four variants of the cryptographic libraries that we evaluated in Section 4.3. The results are summarized in Table 2.

Full-SRC contains the full source code of the libraries, including mpi libraries. ct-verif was only able to analyze BearSSL in this setting (recall that BearSSL has the smallest codebase among all libraries). ct-verif fails with an out-of-memory error on mbedTLS. As for Libgcrypt and OpenSSL, full source introduces a huge amount of source code. Since we have to manually go through the source code line by line with ct-verif to find offending lines. It is a prohibitive amount of work to get all positives. On the contrary, CtChecker could get all four libraries' results, where the result on BearSSL is the same as what ct-verif reports.

No-Undefined-Function corresponds to the minimal source in Table 1. The difference here is that a function call will be removed if it calls an undefined function. The reason is to accommodate the difference in how the tools treat excluded sources. ct-verif treats a return value as sensitive even if there is no tainted argument used in this function call. CtChecker taints all reachable memory in the presence of pointer-values (even if the pointer itself is

---

[4] When verification fails, ct-verif does generate some error messages. However, it is hard to decipher those messages and link them to the source code.

not tainted, see Section 3.2). Removing undefined function calls allows a fair comparison between the two tools. We note that for all libraries, CtChecker is consistently at least as precise as ct-verif, where CtChecker reports fewer positives in 4 out of 7 libraries.

ct-verif-Verified was created from column *No-Undefined-Function* by removing all positives reported by ct-verif, resulting in fully verified code that is constant time. Any positive reported by CtChecker on this version is expected to be a false positive. That being said, CtChecker reported no positive when all positives are removed in ct-verif.

CtChecker-Verified was created similarly from column *No-Undefined-Function* by removing all positives that are reported by CtChecker. Each program is a piece of verified constant-time code. Hence, the positives reported by ct-verif on this version are false positives (we also manually confirmed). ct-verif reports 16 false positives in total on the constant-time code.

To understand the possible causes for these false positives reported by ct-verif, we analyzed its output IR code and results. One reason is that memory nodes within an array are marked public with a constant length by annotation in ct-verif. When a loop is encountered, memory could be accessed with a loop variable. Then, ct-verif fails to determine whether a piece of accessed memory is within the public area or not, because it cannot infer how many iterations at most the loop will be executed. Loop invariants could be automatically computed to verify the range of loop variables. However, the loop invariant generation in ct-verif, based on a simple heuristic, might fail to verify secure code. Another series of false positives originates from how ct-verif handles `memcpy`, for which it will show arbitrary behaviors. For example, in the code snippet below, the addresses of `s`, `p1`, `p2` and `p3` are set to public. The contents of `p1`, `p2` and `p3` are also public.

```
1  int test(int *s, int *p1, int *p2, int *p3) {
2     int a=32;
3     memcpy(p1, p2, a);
4     if (p3[0]) dummy++;
5  }
```

After calling `memcpy`, the content of `p3` is tainted and line 4 will be reported, even though `p3` is neither an argument nor the return of `memcpy`.

In summary, CtChecker exhibited a considerable improvement in precision over ct-verif, a result apparent in the difference between the number of positives reported by each tool in the last three columns. Moreover, as discussed above, CtChecker is more user-friendly as it reports all positives in one shot, whereas using ct-verif to find all positives in source code is cumbersome.

4.6 Comparison with CacheS

CacheS [44] uses abstract interpretation to verify constant-time property. Notably, CacheS is a "soundy" analysis where "the implementation is unsound for speeding up analysis and optimizing memory usage, due to its lightweight but unsound treatment of memory", quoted from the same paper. Also, it operates on a platform independent IR called REIL IR, which is lifted from x86 assembly code. We compare CtChecker against CacheS to show how the memory model and IR code could affect analysis results, see Table 3.

For Libgcrypt, CacheS reports line 19 in Listing 4 (line 682 in *mpi-pow.c*), where `e0` is derived from secret. But CtChecker ignores this line. The reason is that in LLVM IR, this line is neither compiled into a branch nor accesses memory with sensitive index. However, in REIL IR, `cmov` instruction is lifted to a branch with a condition that is derived from the secret, the reason that CacheS reports it.

■ **Table 3** Comparison with CacheS (⋆: low-risk positives; †: extra positive reported by CacheS).

Library	File	CtChecker Positives	CacheS Positives
Libgcrypt 1.10.1	mpi-pow.c	440	440
		559⋆	749⋆
		617	617
		641	–
		667	–
		–	682†
		702⋆	–
mbedTLS 3.2.1	bignum.c	2124	2124
		2127	2127
		2173	2173
		2182⋆	2182⋆
BearSSL 0.6	i32_sub.c	36	N/A
OpenSSL 1.1.1q			
Reciprocal	bn_exp.c	242	N/A
		262	
Mont.		398	
		419	
Mont. Word		1240	

We also observe that CacheS ignores a few high-risk positives reported by CtChecker. In Libgcrypt, two high-risk positives are overlooked, namely, lines 7 and 13 in Listing 4 (lines 641 and 667 in *mpi-pow.c*). In LLVM IR, the branch conditions at these lines are derived from the secret value and they are inside a loop, which leads to multiple-bit leakage. Similar to the previous case, the difference between two tools' analysis targets leads to discrepancies in results. CacheS observes a bsr instruction in the assembly code, which is a constant-time instruction on most architectures.

CacheS employs a lightweight but unsound memory model, which avoids one issue of CtChecker: 69% of false positives of CtChecker are introduced by DSA. However, as a trade-off, this advantage may lead to false negatives in detecting high-risk vulnerabilities, though we did not observe any false negatives from CacheS in our evaluation.

4.7   Comparison with SC-Eliminator's Taint Analysis

As discussed in Section 2.3, SC-Eliminator [46] and CtChecker assume different threat models. As a consequence, SC-Eliminator performs a taint analysis that identifies violations of constant-time disciplines first, before the results are further analyzed by a cache analysis. Here, we compare CtChecker with SC-Eliminator's taint analysis as they both share the same functionality. The comparison is also meaningful as precision improvements in the taint analysis could help cache-analysis tools like SC-Eliminator to rewrite less code, which improves the performance of the product rewritten code.

```
1   count_leading_zeros (c0, e);
2   e = (e << c0);
3   c -= c0;
4   j += c0;
5
6   e0 = (e >> (BITS_PER_MPI_LIMB - W));
7   if (c >= W)
8     c0 = 0;
9   ...
10  count_trailing_zeros (c0, e0);
11  e0 = (e0 >> c0) >> 1;
12
13  for (j += W - c0; j >= 0; j--)
14  {
15    base_u_size = 0;
16    for (k = 0; k < (1<< (W - 1)); k++)
17    {
18      ...
19      base_u_size |= ( precomp_size[k] & (0UL - (k == e0)) );
20    }
21    ...
22  }
```

We build SC-Eliminator from source code[5] with LLVM 8.0.1. The comparison was based on the benchmarks used in [46], see Table 4. Before analyzing the results, we note two major differences between CtChecker and SC-Eliminator's taint analysis:

1. While CtChecker is built on a sound points-to analysis, SC-Eliminator's taint analysis *does not* use any points-to analysis. The result is that the latter might miss taints that are propagated via aliasing.
2. While CtChecker is implemented as an interprocedural analysis, SC-Eliminator's taint analysis is implemented as an *intraprocedural* analysis. Although an intraprocedural analysis is inherently free of context-sensitivity issues that we discuss further in Section 4.8, it requires manual efforts to label sensitive function parameters[6], which is both time-consuming and error-prone.

Despite the differences above, both favor SC-Eliminator's taint analysis in terms of precision, CtChecker reports fewer positives in 6 benchmark programs, while the two tools report the same number of positives on 9 of the benchmark programs. Surprisingly, SC-Eliminator reports less positives in two algorithm programs, namely, *anubis* and *cast5* in the Chronos library. The extra positives that CtChecker reports, as we investigate deeper, are true positives. *But due to the lack of a points-to analysis, SC-Eliminator missed them.* In other words, SC-Eliminator in fact has *false negatives*, which we elaborate next.

4.7.1 False Negatives in SC-Eliminator's Taint Analysis

The lack of a points-to analysis sometimes breaks the propagation of information flow. The code snippet shown in Listing 5 contains a concrete true positive that is missed by SC-Eliminator. Here, the first parameter `key` of function `bar` is the tainted source. At line 8, the first element of `ctx->keys` is tainted by `key`. So, line 9 violates constant-time disciplines

---

[5] https://bitbucket.org/mengwu/timingsyn/src/master/
[6] To reduce the effort, with only a few hard-coded cases, SC-Eliminator's taint analysis assumes that only the first parameter of every function to be tainted.

Table 4 Comparison with SC-Eliminator's Taint Analysis.

Library	SC-Eliminator		CtChecker
	Positives Reported	False Negatives	Positives Reported
appliedCryp/3way.c	4	0	3
appliedCryp/des.c	22	0	18
appliedCryp/loki91.c	8	0	7
chronos/aes.c	388	0	388
chronos/anubis.c	84	8	92
chronos/cast5.c	288	160	448
chronos/cast6.c	448	0	448
chronos/des.c	426	0	416
chronos/des3.c	400	0	390
chronos/fcrypt.c	128	0	128
chronos/khazad.c	40	0	40
libg/camellia.c	32	0	32
libg/des.c	144	0	144
libg/seed.c	8	0	8
libg/twofish.c	248	0	248
supercop/aes__core.c	412	0	409
supercop/cast-ssl.c	448	0	448

as the branch condition is tainted. However, SC-Eliminator misses the positive and leaves it unchanged in the rewritten code. Due to the lack of a points-to analysis, SC-Eliminator cannot infer that variable `keys` defined at line 7 and `ctx->keys` point to the same memory. This contrived example illustrates why SC-Eliminator misses those true positives in *anubis* and *cast5* in the Chronos library.

Listing 5 Example of a false negative in SC-Eliminator.

```
1  void do_something() {...}
2  typedef struct {
3    int **keys;
4    int n;
5  } CONTEXT_st;
6  void bar(int *key, CONTEXT_st *ctx){
7    int **keys = ctx->keys;
8    keys[0] = key;
9    if(ctx->keys[0][0] == 0)
10     do_something();
11 }
```

4.8 Analysis Precision

To answer Q3, we inspected each positive and categorized it into three kinds: low-risk, high-risk and false positive, where the first two are true positives, and their difference is in the severity of information leakage. In particular, a low-risk positive only reveals one bit of information while high-risk positives can leak multiple bits of secrets (e.g., a sensitive branch within a loop).

The classification result is shown in Table 5. CtChecker reports a total number of 724 positives, with 615 true positives and 109 false positives.

True Positives. CtChecker does find true positives in the rewritten code by Constantine. For example, line 5 in Listing 6 is a timing channel and line 17 is a cache side channel. At the beginning, `%idx` is an address calculated from `%sec`, which is derived from a secret

Table 5 Result Classifications. *Base* and *All* refer to the same columns in Table 1.

Library	Base	All	FP	Low-risk	High-risk
Libgcrypt 1.10.1	66	30	24	2	4
mbedTLS 3.2.1	50	10	6	1	3
BearSSL 0.6	18	3	2	0	1
OpenSSL 1.1.1q					
Reciprocal	14	9	7	0	2
Mont.	45	25	23	0	2
Mont. Const. Time	36	18	18	0	0
Mont. Word	4	12	11	0	1
binsec/aes_big	0	0	0	0	0
binsec/des_tab	51	26	2	24	0
binsec/tls-rempad-luk13	7	6	0	6	0
appliedCryp/3way	41	0	0	0	0
appliedCryp/des	72	62	0	62	0
appliedCryp/loki91	75	72	16	56	0
ghostrider/findmax	0	0	0	0	0
ghostrider/matmul	4	0	0	0	0
libg/des	448	432	0	432	0
pycrypto/ARC4	20	19	0	19	0

key. It is then cast to an integer type and is masked by 63. The result `%and` is tainted from the masking operation. At line 4, `%cmp`, the branch condition, is computed from `%and`, making the branch secret dependent. What makes the case more interesting is that the branch at line 5 *does not* exist in the original source code. Constantine adds the branch to check if `%and` satisfies certain property. If not, the execution will stop. For this reason, even though this branch is added into the main processing loop, it should be considered a low-risk positive. Later, `%and` is used to compute another address `%addptr` at line 11. Then, `%addptr` is accessed by `load` if the execution path comes from `forbody.pre`. This memory access is sensitive because the index is derived from secret even after the masking operation. All positives found in rewritten code follow the similar pattern.

Listing 6 Rewritten IR by Constantine from pycrypto/ARC4.

```
1  %idx = gep @stream_state, 0, %sec
2  %2 = ptrtoint %idx to i64
3  %and = and %2, 63
4  %cmp = icmp slt %and, and (sub (add (ptrtoint (@stream_state to i64),
       319), ptrtoint (@stream_state to i64)), -64)
5  br %cmp, label %forbody.pre, label %exit
6
7  exit:
8  ...
9
10 forbody.pre:
11 %addptr = gep @stream_state, 0, %and
12 br label %for.body
13
14 forbody:
15 ...
16 %_ptr = phi [ %addptr1, %forbody ], [ %addptr, %forbody.pre ]
17 %3 = load volatile %_ptr
```

False Positives. While the overall false positive rate of CtChecker on all benchmarks appears low, we further studied the root cause of false positives in the cryptographic library benchmarks, which witness a higher rate of false positives, and found that the majority (63 of them) are caused by the imprecision of DSA, 26 by context-insensitivity, and 2 by

flow-insensitivity (see Section 3.3 for the major reasons of imprecision). Arguably, 69% of false positives due to DSA are unavoidable while we aim for a sound and scalable analysis. For example, when creating the data structure graph for the reciprocal method in OpenSSL, the structure representing the exponent `p` is collapsed. Without the field information, CtChecker have to conservatively taint the whole structure. Consequently, line 1 in Listing 7 (which corresponds to line 171 in `bn_exp.c`) is reported even though it only depends on `flag` field, not the data field.

■ **Listing 7** False positives caused by collapsed memory and callsite-insensitivity in OpenSSL.

```
1  if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0
2    || BN_get_flags(a, BN_FLG_CONSTTIME) != 0
3    || BN_get_flags(m, BN_FLG_CONSTTIME) != 0) {
4    BNerr(BN_F_BN_MOD_EXP_RECP, ERR_R_SHOULD_NOT_HAVE_BEEN_CALLED);
5    return 0;
6  }
```

29% of false positives are produced when a callee function is invoked multiple times within the same caller function with different arguments. In this case, DSA provides no callsite distinction. In Listing 7, `BN_get_flags` is called multiple times with `p`, `a` and `m` as the first parameter, respectively. Since `p` is tainted, lines 2 and 3 (lines 172 and 173 in *bn_exp.c*) are also reported. One fix is to distinguish all calling contexts in the static analysis, or inlining all function calls before the source code is being analyzed. However, both approaches will hurt the scalability of static analysis on large programs.

Other implementations of points-to analyses may be used to improve precision further in two different aspects: (1) to reduce the number of collapsed memory nodes, and (2) to provide finer-grained context-sensitivity on callsites. As an example, these improvements might remove false positives mentioned in the example of reciprocal method in OpenSSL above. However, as developing a more precise points-to analysis is beyond the scope of this paper, we leave it as future work.

The remaining 2 false positives are due to the lack of flow-sensitivity. As discussed in Section 3.2, LLVM's `mem2reg` pass only provides flow-sensitivity to some extent. For example, the tainted data in mbedTLS is the `d` field of variable `E`. Hence, line 1 in Listing 8 (line 1988 in `bignum.c`) is not key-dependent as it only returns the *s* field of *E*.

■ **Listing 8** False positives caused by flow-insensitivity in mbedTLS.

```
1  if( mbedtls_mpi_cmp_int( E, 0 ) < 0 )
2    return( MBEDTLS_ERR_MPI_BAD_INPUT_DATA );
3  ...
4  while( 1 ) // Main processing loop
5  {
6    ...
7    MBEDTLS_MPI_CHK( mpi_select( &WW, W, (size_t) 1 << wsize, wbits ) );
8    ...
9  }
```

However, line 1 is falsely reported by CtChecker. Inside `mpi_select`, `s` field of `WW` is changed to the secret derived from `wbits` that is tainted. Since DSA treats `E` and `WW` as potential aliases, `s` field of `E` is also tainted. However, CtChecker is unable to distinguish *when* `s` field of `E` is tainted. Hence, it conservatively reports line 1 as a positive.

4.9   Scalability

To answer Q4, we evaluate the scalability of CtChecker on benchmark programs from Table 1 on a PC equipped with 2.30GHz Intel Core i7-11800H and 16 GB memory.
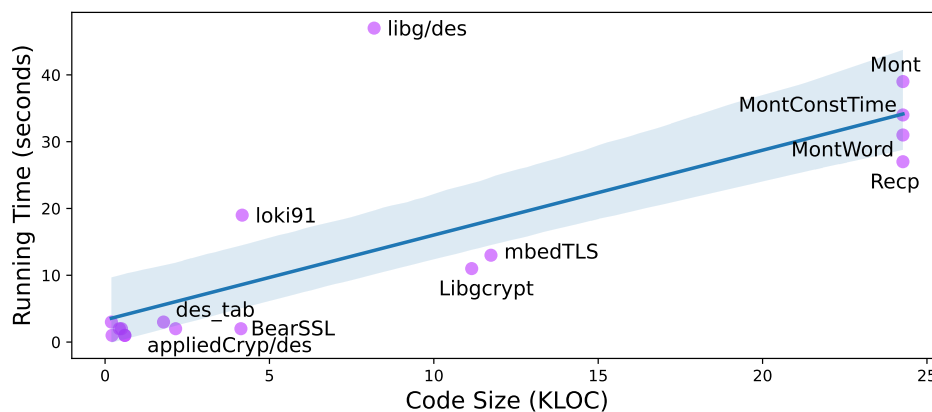
Figure 3 Running Time vs. Code Size of Benchmarks. Unlabeled data points are from Constantine rewritten code with smaller size.

For each cryptographic library, the experiment was done with both the full source (i.e., the SRC version) and the minimal source (i.e., the modular exponentiation code only). Among the tested benchmarks, BearSSL has the smallest codebase of about 4 KLOC, whereas the largest codebase is OpenSSL with around 24 KLOC for its full source version. The running time for these two libraries are 2 and 39 seconds, respectively. The comparison shows that processing time is small even for larger libraries. The libg/des code rewritten by Constantine, which has around 8.6 KLOC, had the longest running time of 48 seconds. Overall, the running time against code size of all benchmark programs being analyzed shows a close to linear trend, as shown in Figure 3.

In terms of memory consumption, the peak usage is around 150 MB when OpenSSL is being analyzed, during the constraint solving step. The result shows that CtChecker is scalable in both the spatial and temporal dimensions.

4.9.1 Running Time of Other Tools

We did not compare the running time and memory usage of CtChecker quantitatively with other tools since the underlying techniques are very different, making a direct comparison unfair. However, we discuss other tools' running time and memory consumption below.

ct-verif: For moderate-sized codebase like mbedTLS, the full source code makes ct-verif run out-of-memory (OOM) after several hours. Moreover, ct-verif stops execution once the first violation is found, making it hard to gauge its execution time if it were reporting all positives in one shot.

CacheS: As reported in [44], the running time for CacheS is at least 33.2 seconds when there is only one function being analyzed, up to 179.2 seconds if the execution runs successfully without timeout, and more than 5 hours if timeout occurs. The memory consumption is also significantly larger than CtChecker, where at least 620 MB of space was used. That said, we note that their reported numbers are collected from different experimental settings than ours, e.g., physical machine, analyzed code base, etc. Therefore, the performance numbers are not directly comparable.

SC-Eliminator: Its taint analysis is the fastest among all tools, which takes around 1 second for each of their benchmarks. The reasons are two-fold however. On the one hand, it does not utilize a points-to analysis, which sacrifices its soundness as we mentioned in Section 4.7.

On the other hand, it simply propagates taints when each instruction is analyzed, which leads to soundness issues in corner cases. For instance, given `x:=y; y:=secret` in a loop, SC-Eliminator fails to taint `x`, as the taint on `y` is discovered later in the analysis. CtChecker is built on more rigorous information flow analysis with constraint generation and solving. The points-to analysis and constraint generation/solving collectively consume most of the time for CtChecker.

Although a fair comparison on efficiency is infeasible, it is safe to conclude that CtChecker's efficiency is better or at least comparable to other tools employing similar sub-components. SC-Eliminator is more efficient than ours and other competitors with a loss of soundness, as discussed above.

5   Related Work

5.1   Detecting Timing Side Channels

Both static and dynamic approaches are widely adopted to detect constant time violations. VirtualCert [3] and FlowTracker [32] are static tools built with formal methods. VirtualCert is flow-insensitive and is specially used for virtualized systems. FlowTracker focuses on optimizing the representation of implicit flows and is flow-sensitive. Almeida et al. [1] propose ct-verif, a static tool that employs self-composition for verifying constant-time property. It either accepts or rejects programs being verified, but it does not pinpoint the source code where the violations occur. Both VirtualCert and ct-verif require additional annotations to work. SecVerilog [51] is a language-based approach for checking hardware-level information flow violations. Somorovsky [37] presents a dynamic tool using fuzzing technique to detect implementation with constant-time violations. Dynamic methods could avoid false positives but are limited by their search space, which leads to unsoundness. Compared to these tools, CtChecker is a sound and generic non-constant-time code detection tool that does not require additional annotations.

5.2   Detecting Cache Side Channels

Cache-based side channels are another type of covert channel that could leak sensitive information to unintended parties. CacheD [45] is a trace-based analysis that identifies cache-based timing channels using taint tracking and symbolic execution. However, symbolic execution might not have the full coverage of execution paths. Brotzman et al. [8] propose a cache-aware symbolic execution (CaSym) that works on LLVM IR. CaSym is able to cover all execution path by introducing a technique that could transform a program with loops to its loop-free version. Both works report the location of vulnerabilities to make it easier for developers to fix them. CacheS [44] is a static analysis that could detect timing channels and cache-based channels. A novel abstract domain called Secret-Augmented Symbolic domain (SAS) is proposed to track sensitive information with high precision while remaining efficient. However, the unsound memory model it uses may cause false negatives. For comparison, CtChecker employs a sound points-to analysis, which makes it both sound and efficient.

5.3   Mitigating Side Channels

Another line of work involves mitigating side channels after vulnerabilities are detected. Cauligi et al. [11] propose a C-like DSL called FaCT. Its compiler is claimed to be able to compile secret-sensitive source code into constant-time LLVM bitcode. However, FaCT

requires libraries to be rebuilt in this language, making it impractical for existing libraries and legacy systems. Wu et al. [46] propose SC-Eliminator, a program rewriter that can eliminate both timing- and cache-based side channels. A constant time select function is proposed for secret-dependent branches. Cache-side channels are removed by preloading all elements in a lookup table. Soares et al. [35] point out that SC-Eliminator introduces out-of-bound memory accesses when doing the transformation. They put forward another rewriter called lif that ensures memory safety at the same time. Preloading methods fail when an attacker could evict cache lines after preloading and before accessing the data. Constantine [7] adopts a radical full linearization design. It focuses on how to maintain efficiency under the radical design. CtChecker does not repair programs when vulnerabilities are found. However, it could help the rewriters to identify problematic code locations more precisely, hence reducing the overhead of mitigation. Moreover, as we demonstrated in the evaluation, it can also serve as an efficient verifier for the code generated by those program rewriters.

6 Conclusion and Future Work

In this work, we build CtChecker, a sound, precise and scalable static information flow analysis for constant-time programming. Compared with traditional information flow analysis, CtChecker is equipped with various features to improve analysis precision on cryptographic code. The features effectively reduce false positive rates while maintaining analysis soundness. By inspecting remaining false positives, we observed that the majority is due to imprecision in the sound points-to analysis that CtChecker is built on.

For future work, a fraction of remaining false positives is due to a callee function being invoked multiple times within the same caller function with different arguments. We plan to investigate how to identify the instances where inlining such code might improve precision, with the insight that heterogeneous arguments to the callee function are the root cause of the imprecision issue. The selective inlining strategy likely will strike a good balance between precision and performance.

References

1 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, pages 53–70, 2016.

2 Lars Ole Andersen. Program analysis and specialization for the c programming language. *Ph.D. Thesis*, 1994.

3 Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1267–1279, New York, NY, USA, 2014. ACM. doi:10.1145/2660267.2660283.

4 Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *International Symposium on Formal Methods*, pages 200–214. Springer, 2011.

5 Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

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

7 Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 715–733, 2021.

8   Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 505–521. IEEE, 2019.

9   Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, and Gang Tan. Specsafe: detecting cache side channels in a speculative world. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–28, 2021.

10   David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

11   Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–189, 2019.

12   GnuPG community. Libgcrypt. `https://gnupg.org/software/libgcrypt/index.html`, 2022.

13   Dorothy E. Denning. A lattice model of secure information flow. *Communication of the ACM*, 19(5):236–243, 1976.

14   William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

15   OpenSSL Software Foundation. Openssl: Cryptography and ssl/tls toolkit. `https://www.openssl.org/`, 2022.

16   Joseph A Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

17   David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

18   Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. *ACM SIGPLAN Notices*, 50(6):291–302, 2015.

19   JuliaHubOSS. Llvm c backend. `https://github.com/JuliaHubOSS/llvm-cbe`, 2018.

20   Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Information Systems Security: 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings 4*, pages 56–70. Springer, 2008.

21   Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

22   Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Notices*, 42(6):278–289, 2007.

23   Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

24   James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4, 2005.

25   Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. *ACM SIGPLAN Notices*, 49(10):525–542, 2014.

26   Colin Percival. Cache missing for fun and profit, 2005.

27   Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure dsa signing exponentiations really are constant-time. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1639–1650, 2016.

28   Thomas Pornin. Constant-time in bearssl. `https://bearssl.org/constanttime.html`, 2018.

29   Thomas Pornin. Bearssl is an implementation of the ssl/tls protocol (rfc 5246) written in c. `https://bearssl.org`, 2022.

30   François Pottier and Vincent Simonet. Information flow inference for ml. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, 2002.

31   Jakob Rehof et al. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2-3):191–221, 1999.

32   Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 110–120, New York, NY, USA, 2016. ACM. `doi:10.1145/2892208.2892230`.

33   Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

34   Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–30, 2020.

35   Luigi Soares and Fernando Magno Quintãn Pereira. Memory-safe elimination of side channels. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 200–210. IEEE, 2021.

36   Juraj Somorovsky. Curious padding oracle in openssl (cve-2016-2107), 2016. Last Retrieved: Jan 2024. URL: `https://web-in-security.blogspot.co.uk/2016/05/curious-padding-oracle-in-openssl-cve.html`.

37   Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1492–1504, New York, NY, USA, 2016. ACM. `doi:10.1145/2976749.2978411`.

38   Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

39   G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 39(11):85–96, 2004.

40   Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

41   TrustedFirmware. Mbed tls. `https://www.trustedfirmware.org/projects/mbed-tls`, 2022.

42   Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *TAP-SOFT'97: Theory and Practice of Software Development: 7th International Joint Conference CAAP/FASE Lille, France, April 14–18, 1997 Proceedings 22*, pages 607–621. Springer, 1997.

43   Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 47(11):2504–2519, 2019.

44   Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying {Cache-Based} side channels through {Secret-Augmented} abstract interpretation. In *28th USENIX security symposium (USENIX security 19)*, pages 657–674, 2019.

45   Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *Proceedings of the 26th USENIX Security Symposium*, pages 235–252, 2017.

46   Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.

47   Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7:99–112, 2017.

48   Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *International Symposium on Formal Methods*, pages 35–51. Springer, 2008.

49   Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Language-based control and mitigation
of timing channels. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming
Language Design and Implementation*, pages 99–110, 2012.

50   Danfeng Zhang and Andrew C Myers. Toward general diagnosis of static errors. In *Proceedings
of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,
pages 569–581, 2014.

51   Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. A hardware design language
for timing-sensitive information-flow security. *ACM SIGPLAN Notices*, 50(4):503–516, 2015.