ABSTRACT
We propose a method, based on program analysis and transformation, for eliminating timing side channels in software code that implements security-critical applications. Our method takes as input the original program together with a list of secret variables (e.g., cryptographic keys, security tokens, or passwords) and returns the transformed program as output. The transformed program is guaranteed to be functionally equivalent to the original program and free of both instruction- and cache-timing side channels. Specifically, we ensure that the number of CPU cycles taken to execute any path is independent of the secret data, and the cache behavior of memory accesses, in terms of hits and misses, is independent of the secret data. We have implemented our method in LLVM and validated its effectiveness on a large set of applications, which are cryptographic libraries with 19,708 lines of C/C++ code in total. Our experiments show the method is both scalable for real applications and effective in eliminating timing side channels.

CCS CONCEPTS
• Security and privacy → Cryptanalysis and other attacks; • Software and its engineering → Compilers; Formal software verification;

KEYWORDS
Side-channel attack, countermeasure, cache, timing, static analysis, abstract interpretation, program synthesis, program repair

1 INTRODUCTION
Side-channel attacks have become increasingly relevant to a wide range of applications in distributed systems, cloud computing and the Internet of things (IoT) where timing characteristics may be exploited by an adversary to deduce information about secret data, including cryptographic keys, security tokens and passwords [24, 53, 54, 61, 67, 83]. Generally speaking, timing side channels exist whenever the time taken to execute a piece of software code depends on the values of secret variables. In this work, we are concerned with two types of timing side-channels: instruction-related and cache-related. By instruction-related timing side channels, we mean the number or type of instructions executed along a path may differ depending on the values of secret variables, leading to differences in the number of CPU cycles. By cache-related timing side channels, we mean the memory subsystem may behave differently depending on the values of secret variables, e.g., a cache hit takes few CPU cycles but a miss takes hundreds of cycles.

Manually analyzing the timing characteristics of software code is difficult because it requires knowledge of not only the application itself but also the micro-architecture of the computer, including the cache configuration and how software code is compiled to machine code. Even if a programmer is able to conduct the aforementioned analysis manually, it would be too labor-intensive and error-prone in practice: with every code change, the software has to be reanalyzed and countermeasure has to be re-applied to ensure a uniform execution time for all possible values of the secret variables. It is also worth noting that straightforward countermeasures such as noise injection (i.e., adding random delay to the execution) do not work well in practice, because noise can be removed using well-established statistical analysis techniques [53, 54].

Thus, we propose an automated method for mitigating timing side channels. Our method relies on static analysis to identify, for a program and a list of secret inputs, the set of variables whose values depend on the secret inputs. To decide if these sensitive program variables lead to timing leaks, we check if they affect unbalanced conditional jumps (instruction-related timing leaks) or accesses of memory blocks spanning across multiple cache lines (cache-related timing leaks). Based on results of this analysis, we perform code transformations to mitigate the leaks, by equalizing the execution time. Although our framework is general enough for a broad range of applications, in this work, we focus on implementing a software tool based on LLVM [6] and evaluating its effectiveness on real cryptographic software.

Figure 1 shows the overall flow of our tool, SC-Eliminator, whose input consists of the program and a list of secret variables. First, we parse the program to construct its intermediate representation inside the LLVM compiler. Then, we conduct a series of static analyses to identify the sensitive variables and timing leaks associated with these variables. Next, we conduct two types of code transformations to remove the leaks. One transformation aims to eliminate the differences in the execution time caused by unbalanced conditional jumps, while the other transformation aims to eliminate the differences in the number of cache hits/misses during the accesses of look-up tables such as S-Boxes.
Conceptually, these transformations are straightforward: If we equalize the execution time of both sensitive conditional statements and sensitive memory accesses, there will be no instruction- or cache-timing leaks. However, since both transformations adversely affect the runtime performance, they must be applied judiciously to remain practical. Thus, a main technical challenge is to develop analysis techniques to decide when these countermeasures are not needed and thus can be skipped safely.

To demonstrate that timing leaks reported by our tool are real and to evaluate the accuracy of our static analyses, we also compile the unmitigated and mitigated software to machine code and carefully analyze the resulting machine code using GEM5 [21], a cycle-accurate micro-architectural CPU simulator. Specifically, given two values of a secret variable, denoted $k_1$ and $k_2$, we first run the original program $P$ and measure the number of CPU cycles spent in each branch. Figure 2 shows the C code of a textbook implementation of a 3-way cipher [76], where the variable $a$ is marked as secret and it affects the execution time of the if-statements. By observing the timing variation, an adversary may be able to obtain information about the bits of $a$.

To remove the dependencies between execution time and secret data, one widely-used approach is equalizing the branches by cross-compiling [7, 56, 65] as illustrated by the code snippet in the middle of Figure 2: the auxiliary variable `dummy_b[3]` and some assignments are added to make both branches contain the same number and type of instructions. Unfortunately, this approach does not always work in practice, due to the presence of hidden states at the micro-architectural levels and related performance optimizations inside modern CPUs (e.g., instruction caching and speculative execution) – we have confirmed this limitation by analyzing the modified code using GEM5, the details of which are described as follows.

We implemented the mitigated program shown in the middle of Figure 2 and, by carefully inspecting the machine code, made sure that all conditional branches indeed had the same number (and type) of instructions. Then, we ran the top-level program on GEM5 with two different cryptographic keys: $k_1$ has 1’s in all 96 bits whereas $k_2$ has 0’s in all 96 bits. Our GEM5 simulation results showed significant timing differences: 88,014 CPU cycles for $k_1$ versus 87,624 CPU cycles for $k_2$. Such timing variation would allow attackers to gain information about the secret key.

Therefore, in the remainder of this paper, we avoid the aforementioned approach while focusing on a safer alternative: replacing sensitive branches with functionally-equivalent, constant-time, and

---

**Figure 1: SC-Eliminator: a tool for detecting and mitigating both instruction- and cache-timing side channels.**
When an index used to access a lookup table (LUT) depends on when all sixteen bytes of \(sbox[256]\) to each original read of \(sbox[0]\), there will be one cache miss followed by fifteen hits; but if \(sbox[0]\) is always accessed, there will be one cache miss followed by fifteen hits.

For example, if we add, as camouflage, accesses of all elements of \(sbox[256]\) on secret data, may affect the execution time. For example, when \(sbox[0]\), as shown in Figure 3, is always ac-

\[
\text{mitigation #1: equalizing the branches}
\]
\[
\text{mitigation #2: removing the branches}
\]

branch-less assignments shown at the bottom of Figure 2. Specifically, \(CTSEL(c,t,e)\) is an LLVM intrinsic we added to ensure the selection of either \(t\) or \(e\), depending on the predicate \(e\), is done in constant time. For different CPU architectures, this intrinsic function will be compiled to different machine codes to obtain the best performance possible (see Section 5 for details). Because of this, our mitigation adds little runtime overhead: the mitigated program requires only 90,844 CPU cycles for both \(k_1\) and \(k_2\).

Note that we cannot simply rely on C-style conditional assignment \(r=(c?t:e)\) because neither \(CTSEL\) nor the LLVM \(select\) instruction because neither guarantees constant-time execution. Indeed, LLVM may transform both to conditional jumps, e.g., when \(r\) is of char type, which may have the same residual timing leaks as before. In contrast, our use of the new \(CTSEL\) intrinsic avoids the problem.

2.2 Table Lookups Affected by Secret Data

When an index used to access a lookup table (LUT) depends on the secret data, the access time may vary due to the behavior of cache associated with the memory block. Such cache-timing leaks have been exploited in block ciphers [44, 67, 80] that, for efficiency reasons, implement S-Boxes using lookup tables. Figure 3 shows the subBytes function of the AES cipher in FELICS [33], which substitutes each byte of the input array (block) with the precomputed byte stored in sbox. Thus, the content of block, which depends on secret data, may affect the execution time. For example, when all sixteen bytes of block are 0x0, meaning sbox[0] is always accessed, there will be one cache miss followed by fifteen hits; but when all sixteen bytes of block differ from each other, there may be 256/64 = 4 cache misses (if we assume 64 bytes per cache line).

Mitigating cache-timing leaks differs from mitigating instruction-timing leaks. Generally speaking, the level of granularity depends on the threat model (i.e., what the attacker can and cannot do). For example, if we add, as camouflage, accesses of all elements of sbox[256] to each original read of sbox[], as shown in Figure 3, it would be impossible for attackers to guess which is the desired element. Since each original loop iteration now triggers the same number of LUT accesses, there is no longer timing variation.

However, the high runtime overhead may be unnecessary, e.g., when attackers cannot observe the timing variation of each loop iteration. If, instead, the attackers can only observe differences in the cache line associated with each write to block[1], it suffices to use the approach in Figure 5. Here, CLS denotes the cache line size (64 bytes in most modern CPUs). Note there is a subtle difference between this approach and the naive preloading (Figure 6): the latter would be vulnerable to Flush+Reload attacks [69, 87]. For example, the attackers can carefully arrange the Flush after Preload is done, and then perform Reload at the end of the victim’s computation; this is possible because Preload triggers frequent memory accesses that are easily identifiable by an attacker. In contrast, the approach illustrated in Figure 5 can avoid such attacks.

If the attackers can only measure the total execution time of a program, our mitigation can be more efficient than Figures 6 and 5. For example, if the cache is large enough to hold all elements, preloading would incur 256/CLS=4 cache misses, but all subsequent accesses would be hits. This approach will be illustrated in Figure 12 (Section 6). However, to safely apply such optimizations, we need to make sure the table elements never get evicted from the cache. For simple loops, this would be easy. But in real applications, loops may be complex, e.g., containing branches, other loops, and function calls, which means in general, a sound static program analysis procedure is needed (see Section 6.2) to determine whether a lookup table access is a MUST-HIT.
We now define the threat model, as well as timing side-channel

to eliminate the timing leaks caused by an idiosyncratic implemen-

tation of the total execution time of the victim’s program with respect
to the secret data. Since this capability is easier to obtain than that
to perform Meltdown/Spectre attacks \[ \text{Evict}+\text{Time}, \text{Prime}+\text{Probe}, \text{and} \text{Flush}+\text{Reload}. \]

We assume \( P \) is a deterministic program whose execution is fixed
completely by the input. Let \( \pi = \text{inst}_1, \ldots, \text{inst}_n \) be an execution
path, and \( \tau(\text{inst}_i) \) is the time taken to execute each instruction
\( \text{inst}_i \), where \( 1 \leq i \leq n \), then, we have \( \tau(\pi) = \sum_{i=1}^{\pi} \tau(\text{inst}_i) \).

Furthermore, \( \tau(\text{inst}_i) \) consists of two components: \( \tau_{\text{cpu}}(\text{inst}_i) \) and
\( \tau_{\text{mem}}(\text{inst}_i) \), where \( \tau_{\text{cpu}} \) denotes the time taken to execute
the instruction itself and \( \tau_{\text{mem}}(\text{inst}_i) \) denotes the time taken to access
the memory. For \( \text{Load} \) and \( \text{Store} \), in particular, \( \tau_{\text{mem}}(\text{inst}_i) \) is determined
by whether the access leads to a cache hit or miss. For the
other instructions, \( \tau_{\text{mem}}(\text{inst}_i) = 0 \). We want to equalize both
components along all program paths – this will be the foundation
of our leak mitigation technique.

4 DETECTING POTENTIAL LEAKS

Now, we present our method for detecting timing leaks, which
is implemented as a sequence of LLVM passes at the IR level. It
takes a set of input variables marked as \textit{secret} and returns a set of
instructions whose execution may depend on these secret inputs.

4.1 Static Sensitivity Analysis

To identify the leaks, we need to know which program variables are
dependent of the \textit{secret} inputs – they are the \textit{sensitive} variables.

Since manual annotation is tedious and error prone, we develop a
procedure to perform such annotation automatically.

\textbf{Secret Source:} The initial set of \textit{sensitive} variables consists
of the secret inputs marked by the user. For example, in a block cipher,
the secret input would be the cryptographic key while the plaintext
would be considered as public.

\textbf{Tag Propagation:} The \textit{sensitivity} tag is an attribute to be propa-
gated from the secret source to other program variables following
either data- or control-dependency transitively. An example of data-
dependency is the \textit{def-use} relation in \( (b = a \& 0x80; \) where \( b \) is
marked as sensitive because it depends on the most significant bit
of \( a \), the sensitive variable. An example of control-dependency is
\( \text{if}(a=0x10) \{ b=1; \} \text{ else } \{ b=0; \} \) where \( b \) is marked as sensitive
because it depends on whether \( a \) is \( 0x10 \).

\textbf{Field-sensitivity Analysis:} To perform the static analysis defined
above, we need to identify aliased expressions, \textit{e.g.}, syntactically-
different variables or fields of structures that point to the same
memory location. Cryptographic software code often has this type
of pointers and structures. For example, the ASE implementation
of Chronos \[32\] shown in Figure 8 demonstrates the need for
field-sensitivity during static analysis. Here, local pointer key be-
comes sensitive when \( \text{key}[0] \) is assigned the value of another sen-
tive variable in \textit{key}. Without field sensitivity, one would have to
mark the entire structure as sensitive to avoid missing potential
leaks. In contrast, our method performs a field-sensitive pointer
analysis \[15, 71\] to propagate the sensitivity tag only to relevant
fields such as \textit{key\_enc} inside \textit{ctx}, while avoiding fields such as
\textit{key\_length}. This means we can avoid marking (false) the unbal-
anced \textit{if} \( (\text{ctx}\rightarrow\text{key\_length}) \) statement as leaky.

typedef struct {
uint32_t *sk; // the round keys
int nr; // the number of rounds
} rc5_ctx;
#define ROTL32(X,C) (((X)<<(C))|((X)>>(32-(C))))
void rc5_encrypt(rc5_ctx *c, uint32_t *data, int blocks) {
uint32_t *d,*sk;
typedef struct {
uint32_t x[32];
uint32_t r[32];
uint32_t xk[32];
extern void AES_set_encrypt_key(uint32_t *key, int key_length);
} rc5_ctx;
int h,c;
d = data;
if (c>0) {
for (h=0; h<blocks; h++) {
d[0] = c^=sk[0];
d[1] = c^=sk[1];
for (i=0; i<nr*2; i+=2) {
d[0] ^= d[1];
r = d[1] & 3;
d[0] = ROTL32(d[0],c);
d[1] = ROTL32(d[1],c);
}
}
}

Figure 7: Code snippet from RC5.c

2.3 Idiosyncratic Code Affected by Secret Data

For various reasons, certain operations in cryptographic software
are often implemented using a series of simpler but functionally-
equivalent operations. For example, the shift operation \( X<<C \)
may be implemented using a sensitive data-dependent loop with
additions: \( \text{for}(i=0; i<C; i++) \{ X = X; \} \) because some targets
\textit{e.g.} MSP430 do not support multi-bit shifts.

One real example of such idiosyncratic code is the implemen-
tation of \texttt{rc5\_encrypt} \[76\] shown in Figure 7. Here, the second
parameter of \texttt{ROT32()} is aliased to the sensitive variable \texttt{c->sk}.

To eliminate the timing leaks caused by an idiosyncratic implemen-
tation of \( X<<C \), we must conservatively estimate the loop bound.
If we know, for example, the maximum value of \( C \) is \texttt{MAX\_C}, the data-
dependent loop may be rewritten to one with a fixed loop bound:
\( \text{for}(i=0; i<\text{MAX\_C}; i++) \{ \text{if}(i<C) \ X = X; \} \). After this transformation,
we can leverage the aforementioned mitigation techniques to eliminate
leaks associated with the \textit{if}(i<C) statement.

3 THREAT MODEL

We now define the threat model, as well as timing side-channel
leaks under our threat model.

We assume a \textit{less-capable} attacker who can only observe varia-
tion of the total execution time of the victim’s program with respect
to the secret data. Since this capability is easier to obtain than that
of a \textit{more-capable} attacker, it will be more widely applicable.
A classic example, for instance, is when the victim’s program runs
on a server that can be probed and timed remotely by the attacker
using a malicious client.

We do not consider the \textit{more-capable} attacker who can directly
access the victim’s computer to observe hidden states of the CPU
at the micro-architectural levels, \textit{e.g.}, by running malicious code
to perform Meltdown/Spectre attacks \[52, 59\] or similar cache at-
tacks \[69, 87\] (\textit{Evict\_Time}, \textit{Prime\_Probe}, and \textit{Flush\_Reload}). Mitигating
such attacks at the software level only will likely be signifi-
cantly more expensive — we leave it for future work.

Let \( P \) be a program and \( in = (X,K) \) be the input, where \( X \) is
\textit{public} and \( K \) is \textit{secret}. Let \( x \) and \( k \) be concrete values of \( X \) and \( K \),
respectively, and \( \tau(P,x,k) \) be the time taken to execute \( P \) under \( x \)
and \( k \). We say \( P \) is free of timing side-channel leaks if
\[ \forall x,k_1,k_2 : \tau(P,x,k_1) = \tau(P,x,k_2) . \]

That is, the execution time of \( P \) is independent of the secret input \( K \).
When \( P \) has timing leaks, on the other hand, there must exist some
\( x, k_1 \) and \( k_2 \) such that \( \tau(P,x,k_1) \neq \tau(P,x,k_2) \).

We assume \( P \) is a deterministic program whose execution is fixed
completely by the input. Let \( \pi = \text{inst}_1, \ldots, \text{inst}_n \) be an execution
path, and \( \tau(\text{inst}_i) \) be the time taken to execute each instruction
\( \text{inst}_i \), where \( 1 \leq i \leq n \); then, we have \( \tau(\pi) = \sum_{i=1}^{\pi} \tau(\text{inst}_i) \).

Furthermore, \( \tau(\text{inst}_i) \) consists of two components: \( \tau_{\text{cpu}}(\text{inst}_i) \) and
\( \tau_{\text{mem}}(\text{inst}_i) \), where \( \tau_{\text{cpu}} \) denotes the time taken to execute
the instruction itself and \( \tau_{\text{mem}}(\text{inst}_i) \) denotes the time taken to access
the memory. For \( \text{Load} \) and \( \text{Store} \), in particular, \( \tau_{\text{mem}}(\text{inst}_i) \) is determined
by whether the access leads to a cache hit or miss. For
the other instructions, \( \tau_{\text{mem}}(\text{inst}_i) = 0 \). We want to equalize both
components along all program paths – this will be the foundation
of our leak mitigation technique.

Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang
There are two requirements for a branch statement to have potential timing leaks. First, the condition depends on secret data. Second, the branches are unbalanced. Figure 2 shows an example, where the conditions depend on the secret input and the branches obviously are unbalanced. Sometimes, however, even if two conditional branches have the same number and type of instructions, they still result in different execution time due to hidden micro-architectural states, as we have explained in Section 2 and confirmed using GEM5 simulation. Thus, to be conservative, we consider all sensitive conditional statements as potential leaks (regardless of whether they are balanced) and apply our CTSEL based mitigation.

4.3 Leaky Lookup-table Accesses

The condition for a lookup-table (LUT) access to leak timing information is that the index used in the access is sensitive. In practice, the index affected by secret data may cause memory accesses to be mapped to different cache lines, some of which may have been loaded and thus result in hits while others result in misses. Therefore, we consider LUT accesses indexed by sensitive variables as potential leaks, e.g., the load from sbox in Figure 3, which is indexed by a sensitive element of block.

However, not all LUT accesses are leaks. For example, if the table has already been loaded, the (sensitive) index would no longer cause differences in the cache. This is an important optimization we perform during mitigation — the analysis required for deciding if an LUT access results in a must-hit will be presented in Section 6.2.

5 MITIGATING CONDITIONAL STATEMENTS

In this section, we present our method for mitigating leaks associated with conditional jumps. In contrast to existing approaches that only attempt to balance the branches, e.g., by adding dummy instructions [7, 56, 65], we eliminate these branches.
the branches before their merge point. The pseudo code, shown in Algorithm \ref{alg:mitigatebranch}, takes the entry block \textit{bb} as input.

**Condition and CTSEL**: First, we assume the existence of CTSEL\((c,t,e)\), a constant-time intrinsic function that returns \(t\) when \(c\) equals \textit{true}, and \(e\) when \(c\) equals \textit{false}. Without any target-specific optimization, it may be implemented using bit-wise operations: CTSEL\((c,t,e)\) \(\{c0=c<1; \ c1=c\&c<1; \ v=\ (c0 \& e)(c1 \& t)\}\) when the variables are of ‘char’ type and \(c\) is \textit{true}, \(c0\) will be 0x00 and \(c1\) will be 0xFF; and when \(c\) is \textit{false}, \(c0\) will be 0xFF and \(c1\) will be 0x00. With target-specific optimization, CTSEL\((c,t,e)\) may be implemented more efficiently. For example, on x86 or ARM CPUs, we may use CMOVCC instructions as follows: \{(MOV val t; CMP c 0x0; CMOVQ val e;\} which requires only three instructions. We will demonstrate through experiments (Section 7) that target-specific optimization reduces the runtime overhead significantly.

**Algorithm 2: Mitigating the conditional statement from \textit{bb}.**

```
1 MitigateBranch(BasicBlock \textit{bb})
2 begin
3   Let \textit{cond} be the branch condition associated with \textit{bb};
4   foreach Instruction \textit{i} in \text{THEN} branch or \text{ELSE} branch do
5     if \textit{i} is a \textit{Store} of the value \textit{val} to the memory address \textit{addr} then
6        Let v’\textit{val} = CTSEL(\textit{cond}, \textit{val}, \text{Load}(\textit{addr}));
7        Replace \textit{i} with the new instruction \textit{Store}(v’\textit{val}, \textit{addr});
8   end
9   foreach Phi Node (\textit{rv} = \phi(\textit{sr1}, \textit{sr2}, \ldots)) at the merge point do
10      Let v’\textit{rv} = CTSEL(\textit{cond}, \textit{sr1}, \textit{sr2});
11      Replace the Phi Node with the new instruction (\textit{rv} = v’\textit{rv});
12   end
13   Change the conditional jump to \text{THEN} branch to unconditional jump;
14   Delete the conditional jump to \text{ELSE} branch;
15   Redirect the outgoing edge of \text{THEN} branch to start of \text{ELSE} branch;
16 end
```

**Store Instructions**: Next, we transform the branches. If the instruction is a Store\((val,addr)\) we replace it with CTSEL. That is, the Store instructions in \text{THEN} branch will only take effect when the condition is evaluated to \textit{true}, while the Store instructions in \text{ELSE} branch will only take effect when condition is \textit{false}.

**Local Assignments**: The above transformation is only for memory Store\((val,addr)\) to a register variable such as if \textit{cond} \{\textit{rv}=val1; \ldots\} else \{\textit{rv}=val2; \ldots\} because, inside LLVM, the latter is represented in the static single assignment (SSA) format. Since SSA ensures each variable is assigned only once, it is equal to if \textit{cond} \{\%rv1=val1; \ldots\} else \{\%rv2=val2; \ldots\} together with a Phi Node added to the merge point of these branches.

**The Phi Nodes**: The Phi nodes are data structures used by compilers to represent all possible values of local (register) variables at the merge point of CFG paths. For \%rv \(\leftarrow \phi(\%rv1, \%rv2)\), the variables \%rvT and \%rvE in SSA format denote the last definitions of \%rv in the \text{THEN} and \text{ELSE} branches: depending on the condition, \%rv gets either \%rvT or \%rvE. Therefore, in our procedure, for each Phi node at the merge point, we create an assignment from the newly created \textit{val’} to \%rv, where \textit{val’} is again computed using CTSEL.

**Unconditional Jumps**: After mitigating both branches and the merge point, we can eliminate the conditional jumps with unconditional jumps. For the standardized branches on the left-hand side of Figure 11, the transformed CFG is shown on the right-hand side.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure11.png}
\caption{Removing the conditional jumps.}
\end{figure}

\subsection{5.3 Optimizations}

The approach presented so far still has redundancy. For example, given if\{(\textit{cond}) \{\textit{addr}=val1;\} else \{\textit{addr}=val2;\}\} the transformed code would be \{(\textit{addr} = CTSEL\((\textit{cond}, \textit{val1}, \text{\textit{addr}})\); \textit{addr} = CTSEL\((\textit{cond}, \textit{addr}, \textit{val2})\))\} which has two CTSEL instances. We can remove one or both CTSEL instances:

- If \{(\textit{valp}=val1)\} holds, we merge the two Store operations into one Store: \textit{addr} = val1.
- Otherwise, we use \textit{addr} = CTSEL\((\textit{cond}, \textit{val1}, \textit{val2})\).

In the first case, all CTSEL instances are avoided. Even in the second case, the number of CTSEL instances is reduced by half.

\section{6 MITIGATING LOOKUP-TABLE ACCESSES}

In this section, we present our method for mitigating lookup-table accesses that may lead to cache-timing leaks. In cryptographic software, such leaks are often due to dependencies between indices used to access S-Boxes and the secret data. However, before delving into the details of our method, we perform a theoretical analysis of the runtime overhead of various alternatives, including even those designed against the more-capable attackers.

\subsection{6.1 Mitigation Granularity and Overhead}

We focus primarily on less-capable attackers who only observe the total execution time of the victim’s program. Under this threat model, we develop optimizations to take advantage of the cache structure and unique characteristics of the software being protected. Our mitigation, illustrated by the example in Figure 12, can be significantly more efficient than the approaches illustrated in Figure 5.

In contrast, the Byte-access-aware threat model allows attackers to observe timing characteristics of each instruction in the victim’s program, which means mitigation would have to be applied to every LUT access to make sure there is no timing difference (Figure 4). The Line-access-aware threat model allows attackers to see the difference between memory locations mapped to different cache lines. Thus, mitigation only needs to touch all cache lines associated with the table (Figure 5).

Let \(P\) be a path in \(P\) and \(r(\pi)\) be its execution time. Let \(t_{max}\) be the maximum value of \(r(\pi)\) for all possible \(\pi\) in \(P\). For our Total-time-aware threat model, the ideal mitigation would be a program \(P’\) whose execution time along all paths matches \(t_{max}\). In this case, we say mitigation has no additional overhead. We quantify the overhead of other approaches by comparing to \(t_{max}\).

Table 1 summarizes the comparison. Let \(N\) be the table size, \(CLS\) be the cache line size, and \(M = \lceil N/CLS \rceil\) be the number of
Table 1: Overhead comparison: \( N \) is the table size; \( M = \lceil N/CLS \rceil \) is the number of cache lines to store the table; \( K \) is the number of times table elements are accessed.

<table>
<thead>
<tr>
<th>Program Version</th>
<th># Accesses</th>
<th># Cache Misses</th>
<th># Cache Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original program</td>
<td>( K ) from ( M ) to ( 1 ) from ( K ) to ( K-1 )</td>
<td>( K )</td>
<td>( K )</td>
</tr>
<tr>
<td>Granularity: Byte-access</td>
<td>( K )</td>
<td>( M )</td>
<td>( K )</td>
</tr>
<tr>
<td>Granularity: Line-access</td>
<td>( K )</td>
<td>( M )</td>
<td>( K )</td>
</tr>
<tr>
<td>Granularity: total time ( t_{\text{max}} )</td>
<td>( K )</td>
<td>( M )</td>
<td>( K )</td>
</tr>
<tr>
<td>Our Method: opt. w/ cache analysis</td>
<td>( K )</td>
<td>( M )</td>
<td>( K-1 )</td>
</tr>
</tbody>
</table>

Figure 12: Reduction: preloading only in the first iteration.

cache lines needed. Let \( K \) be the number of times table elements are accessed. Without loss of generality, we assume each element occupies one byte. In the best case where all \( K \) accesses are mapped to the same cache line, there will be 1 miss followed by \( K-1 \) hits. In the worst case \( t_{\text{max}} \) where the \( K \) accesses are scattered in \( M \) cache lines, there will be \( M \) misses followed by \( K-M \) hits.

When migrating at the granularity of a byte (e.g., Figure 4), the total number of accesses in \( P' \) is increased from \( K \) to \( K+N \). Since all elements of the table are touched when any element is read, all cache lines will be accessed. Thus, there are \( M \) cache misses followed by \( K-N-M \) hits.

When migrating at the granularity of a line (e.g., Figure 5), the total number of accesses becomes \( K+N \). Since all cache lines are touched, there are \( M \) cache misses followed by \( K-M \) hits.

Our method, when equipped with static cache analysis-based optimization (Section 6.2), further reduces the overhead: by checking whether the table, once loaded to the cache, will stay there until all accesses complete. If we can prove the table never gets evicted, we only need to load each line once. Consequently, there will be \( M \) misses in the first loop iteration, followed by \( K-1 \) hits in the remaining \( K-1 \) loop iterations.

In all cases, however, the number of cache misses \( (M) \) matches that of the ideal mitigation; the differences is only in the number of cache hits, which increases from \( K-N \) to \( K+N-M \), \( K-M \), or \( K \). Although these numbers (of hits) may differ significantly, the actual time difference may not be, because a cache hit often takes an order of magnitude shorter time than a cache miss.

6.2 Static Cache Analysis-based Reduction

We develop a static cache analysis to compute, at any location, whether a memory element is definitely in the cache. This MUST-HIT analysis allows us to decide if an LUT access needs mitigation. For example, in subCell() of LED_encrypt, \( c \) that accesses sbox[16] using for (i=0; i<4; i++) for (j=0; j<4; j++) {state[i][j]=sbox[state[i][j]]}; since the size of sbox is 16 bytes while a cache line has 64 bytes, all the elements can be stored in the same cache line. Therefore, the first loop iteration would have a cache miss while all subsequent fifteen iterations would be hits—there is no cache-timing leak that needs mitigation.

There are many other applications where lookup-table accesses result in MUST-HITs, e.g., block ciphers with multiple encryption or decryption rounds, each of which accesses the same lookup table. Instead of mitigating every round, we use our cache analysis to check if, starting from the second round, mitigation can be skipped.

Abstract Domain. We design our static analysis procedure based on the unified framework of abstract interpretation \( [30, 40, 41] \), which defines a suitable abstraction of the program’s state as well as transfer functions of all program statements. There are two reasons for using abstract interpretation. The first one is to ensure the analysis can be performed in finite time even if precise analysis of the program may be undecidable. The second one is to summarize the analysis results along all paths and for all inputs.

Without loss of generality, we assume the cache has full associativity and a set \( L = \{l_1, ..., l_N\} \) of cache lines. The subscript of \( l_i \), where \( 1 \leq i \leq N \), denotes the age: \( 1 \) is the youngest, \( N \) is the oldest, and \( > N \) means the line is outside of the cache. For ease of presentation, let \( l_x \) be the imaginary line outside of the cache. Thus, \( L = L \cup \{l_x\} \) is the extended set of cache lines.

Let \( V = \{v_1, ..., v_n\} \) be the set of program variables, each of which is mapped to a subset \( L \subseteq L' \) of cache lines. The age of \( v \) in \( V \), denoted \( \text{Age}(v) \), is a set of integers corresponding to ages (subscripts) of the lines it may reside on (all paths and for all inputs). The program’s cache state, denoted \( S = (\text{Age}(v_1), ..., \text{Age}(v_n)) \), provides the ages of all variables.

Consider an example program with three variables \( x, y \) and \( z \), where \( x \) is mapped to the first cache line, \( y \) may be mapped to the first two lines (e.g., along two paths) and \( z \) may be mapped to Lines 3-5. Thus, \( L_x = \{l_1\}, L_y = \{l_1, l_2\} \), and \( L_z = \{l_1, l_3, l_5\} \), and the cache state is \( (\{1\}, \{1, 2\}, \{3, 4, 5\}) \).

Transfer Functions. The transfer function of each program statement defines how it transforms the cache state to a new state. Without loss of generality, we assume the cache uses the popular least recent used (LRU) update policy: Recall that in a fully associative cache, a memory block may be mapped to any cache line; and under LRU, the cache keeps the most recently used memory blocks while evicting the least recently used blocks.

Figure 13 shows two examples. On the left-hand side, the initial state, for variables \( a, b, c, d, e \), is \( (\{1\}, \{2\}, \{3\}, \{4\}, \{5\}) \). After accessing \( e \), the new state is \( (\{2\}, \{3\}, \{4\}, \{5\}, \{1\}) \). On the right-hand side, the initial state is \( (\{1\}, \{2\}, \{3\}, \{4\}, \{5\}) \). After accessing \( e \), the new state is \( (\{2\}, \{3\}, \{4\}, \{5\}, \{1\}) \). In both cases, the newly accessed \( e \) gets the youngest age, while the ages of other blocks either decrement or remain the same. Since \( d \) is the oldest block (age 5), it stays outside of the cache.

The transfer function TFunc\( (S, \text{inst}) \) models the impact of instruction \( \text{inst} \) on the cache state \( S \); it returns a new cache state
sbox has four elements in total. In the original state, the first three
ke (meant to be) complete in finding all MUST-HIT cases – insisting
Therefore, when it says the variable is in the cache, it is guaranteed
Thus, the function TFunc models what is illustrated in Figure 13.
MUST-HIT Analysis. Since our goal is to decide whether a memory block is definitely in the cache, we compute in Age(v) the upper bound of all possible ages of v, e.g., along all paths and for all inputs. If this upper bound is ≤ N, we know v must be in the cache.
We also define the join (∪) operator accordingly: it is needed to merge states S and S′ from different paths. It is similar to set intersection—in the resulting S″ = S ∪ S′, each Age′′(v) gets the maximum of Age(v) in state S and Age′(v) in state S′. This is because v ∈ V is definitely in the cache only if it is in the cache according to both states, i.e., Age′(v) ≤ N and Age′′(v) ≤ N.
Consider the left example in Figure 14, where the ages of a are 1 and 3 before reaching the merge point, and the ages of c are 3 and 2. After joining the two cache states, the ages of a and c become 3, and the age of d remains 4. The ages of b and e become ∆ because, in at least one of the two states, they are outside of the cache.
Now, consider the right-hand-side example in Figure 14, where sbox has four elements in total. In the original state, the first three elements are in the cache whereas sbox[3] is outside. After accessing sbox[key], where the value of key cannot be statically determined, we have to assume the worst case. In our MUST-HIT analysis, the worst case means key may be any index ranging from 0 to 3. To be safe, we assume sbox[key] is mapped to the oldest element sbox[3]. Thus, the new state has sbox[3] in the first line while the ages of all other elements are decremented.
Correctness and Termination. Our analysis is a conservative approximation of the actual cache behavior. For example, when it says a variable has age 2, its actual age must not be older than 2. Therefore, when it says the variable is in the cache, it is guaranteed to be in the cache, i.e., our analysis is sound; however, it is not (meant to be) complete in finding all MUST-HIT cases—insisting on being both sound and complete could make the problem undecidable. In contrast, by ensuring the abstract domain is finite (with finitely many lines in L and variables in V) and both TFunc and ∪ are monotonic, we guarantee that our analysis terminates.
Handling Loops. One advantage of abstract interpretation [30, 40, 41] is the capability of handling loops: for each back edge in the CFG, cache states from all incoming edges are merged using the join (∪) operator. Nevertheless, loops in cryptographic software have unique characteristics. For example, most of them have fixed loop bounds, and many are in functions that are invoked in multiple encryption/decryption rounds. Thus, memory accesses often cause cache misses in the first loop iteration of the first function invocation, but hits subsequently. Such first-miss followed by always hit, however, cannot be directly classified as a MUST-HIT.
To exploit the aforementioned characteristics, we perform a code transformation prior to our analysis: unrolling the first iteration out of the loop while keeping the remaining iterations. For example, for(i=0;i<16;++i) {block[i]=...} becomes for(i=1; i<16;++i) {block[i]=...}. As soon as accesses in the first iteration are mitigated, e.g., as in Figure 12, all subsequent loop iterations will result in MUST-HITS, meaning we can skip the mitigation and avoid the runtime overhead. Our experiments on a large number of real applications show that the cache behaviors of many loops can be exploited in this manner.

7 EXPERIMENTS
We have implemented our method in a tool named SC-Eliminator, which takes LLVM bit-code as input and returns leak-free bit-code as output. The new bit-code may be compiled to machine code to run on any platform (e.g., x86 and ARM) using standard tools or simulated by GEM5 to obtain timing statistics.
We conducted experiments on C/C++ programs that implement well-known cryptographic algorithms by compiling them to bit-code using Clang/LLVM. Table 2 shows the benchmark statistics. In total, there are 19,708 lines of code from libraries including a real-time Linux kernel (Chronos [32]), a lightweight cryptographic library (FELICS [2]), a system for performance evaluation of cryptographic primitives (SuperCop [5]), the Botan cryptographic library [1], three textbook implementations of cryptographic algorithms [76], and the GNU Libgcrypt library [4]. Columns 1 and 2 show the benchmark name and source. Column 3 shows the number of lines of code (LoC). Columns 4 and 5 show the number of conditional jumps (# IF) and the number of lookup tables (# LUT). The last two columns show more details of these lookup tables, including the total, minimum, and maximum table sizes.

Table 2: Benchmark statistics.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>LoC</th>
<th># IF</th>
<th># LUT</th>
<th>LUT size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>AES in Chronos [32]</td>
<td>1099</td>
<td>16</td>
<td>8,488</td>
<td>(40, 1024)</td>
</tr>
<tr>
<td>cast</td>
<td>CAST in SuperCop [3]</td>
<td>942</td>
<td>5</td>
<td>16,384</td>
<td>(2048, 2048)</td>
</tr>
<tr>
<td>fcrypt</td>
<td>FCrypt encryption in Chronos [32]</td>
<td>1453</td>
<td>20</td>
<td>8,104</td>
<td>(64, 1024)</td>
</tr>
<tr>
<td>cast128</td>
<td>cast 128-bit in Botan [1]</td>
<td>217</td>
<td>2</td>
<td>1,005</td>
<td>(64, 1024)</td>
</tr>
<tr>
<td>des</td>
<td>des cipher in Botan [1]</td>
<td>855</td>
<td>12</td>
<td>10,240</td>
<td>(2048, 2048)</td>
</tr>
<tr>
<td>kasumi</td>
<td>kasumi cipher in Botan [1]</td>
<td>275</td>
<td>2</td>
<td>1,152</td>
<td>(128, 2048)</td>
</tr>
<tr>
<td>seed</td>
<td>seed cipher in Botan [1]</td>
<td>352</td>
<td>5</td>
<td>4,160</td>
<td>(64, 1024)</td>
</tr>
<tr>
<td>twofish</td>
<td>twofish cipher in Botan [5]</td>
<td>729</td>
<td>18</td>
<td>5,150</td>
<td>(128, 2048)</td>
</tr>
<tr>
<td>sky</td>
<td>sky cipher reference [7]</td>
<td>179</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>loki91</td>
<td>loki cipher reference [76]</td>
<td>231</td>
<td>10</td>
<td>32</td>
<td>(12, 32)</td>
</tr>
<tr>
<td>camellia</td>
<td>camellia cipher in Libgcrypt [4]</td>
<td>1488</td>
<td>15</td>
<td>4,096</td>
<td>(1024, 4096)</td>
</tr>
<tr>
<td>seed</td>
<td>seed cipher in Libgcrypt [4]</td>
<td>488</td>
<td>5</td>
<td>4,160</td>
<td>(64, 1024)</td>
</tr>
<tr>
<td>twofish</td>
<td>twofish cipher in Libgcrypt [4]</td>
<td>1899</td>
<td>1</td>
<td>6,380</td>
<td>(256, 4096)</td>
</tr>
</tbody>
</table>
Table 3: Results of conducting static leak detection.

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>Sensitive (leaks)</th>
</tr>
</thead>
<tbody>
<tr>
<td># IF</td>
<td># LUT</td>
<td># LUT-access</td>
</tr>
<tr>
<td># IF</td>
<td># LUT</td>
<td># LUT-access</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ase</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>des</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>des3</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>anubis</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>cast5</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>cast6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>crypt</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>khazad</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>71Block</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Twofish</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>ase</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>cast</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>cast128</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>des</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>kasumi</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>seed</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>twofish</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>ase</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>cast</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>cast128</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>des</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>kasumi</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>seed</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>twofish</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>ase</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>cast</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>cast128</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>des</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>kasumi</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>seed</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>twofish</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>ase</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>cast</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>cast128</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>des</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>kasumi</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>seed</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>twofish</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

Our experiments aimed to answer three research questions: (1) Is our method effective in mitigating instruction- and cache-timing leaks? (2) Is our method efficient in handling real applications? (3) Is the overhead of the mitigated code, in terms of code size and run time, low enough for practical use?

7.1 Results: Leak Detection

Table 5 shows the results of applying our leak detection technique, where Columns 1-4 show the name of the benchmark together with the number of conditional jumps (# IF), lookup tables (# LUT), and accesses to table elements (# LUT-access), respectively. Columns 5-7 show the number of sensitive conditional jumps, lookup tables, and accesses, respectively. Thus, non-zero in the sensitive LUT column means there is instruction-timing leakage, and non-zero in the sensitive LUT-access means there is cache-timing leakage. We omit the time taken by our static analysis since it is negligible: in all cases the analysis completed in a few seconds.

Although conditional statements (#IF) exist, few are sensitive. Indeed, only twofish from Botan[] and three old textbook implementations (3way, des, and loki91) have leaks of this type. In contrast, many lookup tables are sensitive due to cache. This result was obtained using fully associative LRU cache with 512 cache lines, 64 bytes per line, and thus 32 Kilobytes in total.

Some benchmarks, e.g., aes.key from Botan[], already preload lookup tables; however, our analysis still reports timing leakage, as shown in Figure 15, where XEK is key-related and used to access an array in the second for-loop. Although the table named TD is computed at the run time (thus capable of avoiding flush+reload attack) and all other tables are preloaded before accesses, it forgot to preload SE[256], which caused the cache-timing leak.

7.2 Results: Leak Mitigation

To evaluate whether our method can robustly handle real applications, we collected results of applying our mitigation procedure to each benchmark. Table 4 shows the results. Specifically, Columns 2-5 show the result of our mitigation without cache analysis-based optimization, while Columns 6-9 show the result with the optimization. In each case, we report the number of LUT accesses actually mitigated, the time taken to complete the mitigation, the increase in program size, and the increase in runtime overhead. For anubis, in particular, our cache analysis showed that only 10 out of the 868 sensitive LUT accesses needed mitigation; as a result, optimization reduced both the program’s size (from 9.08x to 1.10x) and its execution time (from 6.90x to 1.07x).

We also compared the execution time with generic (bitwise) versus optimized (CMOV) implementations of CTSIEL(c, t, e). Figure 16 shows the result in a scatter plot, where points below the diagonal line are cases where the optimized implementation is faster.

7.3 Results: GEM5 Simulation

Although our analysis is conservative in that the mitigated code is guaranteed to be leak-free, it is still useful to conduct GEMS simulations, for two reasons. First, it confirms our analysis reflects the reality: the reported leaks are real. Second, it demonstrates that, after mitigation, leaks are indeed eliminated.

Table 5 shows our results. For each benchmark, we ran the machine code compiled for x86 on GEMS using two manually crafted inputs (e.g., cryptographic keys) capable of showing the timing variation. Columns 2-5 show the results of the original program,
including the number of CPU cycles taken to execute it under the two inputs, as well as the number of cache misses. Columns 6-9 show the results on the mitigated program versions.

The results show the execution time of the original program indeed varies, indicating there are leaks. But it becomes constant after mitigation, indicating the leaks are removed. The one exception is aes_key: we were not able to manually craft the inputs under which leak is demonstrable on GEM5. Since the input space is large, manually crafting such inputs is not always easy: symbolic execution tools [45–48] may help generate such leak-manifesting input pairs — we will consider it for future work.

7.4 Threats to Validity
First, our mitigation is software based; as such, we do not address leaks exploitable only by probing the hardware such as instruction pipelines and data buses. We focus on the total-time-aware threat model: although extensions to handle other threat models are possible (e.g., multi-core and multi-level cache), we consider them as future work. It is possible that timing characteristics of the machine code may differ from those of the LLVM bit-code, but we have taken efforts in making sure machine code produced by our tool does not deviate from the mitigated bit-code. For example, we always align sensitive lookup tables to cache line boundaries, and we implement CTSEL as an intrinsic function to ensure constant-time execution. We also use GEM5 simulation to confirm that machine code produced by our tool is indeed free of timing leaks.

8 RELATED WORK
Kocher [53] is perhaps the first to publicly demonstrate the feasibility of timing side-channel attacks in embedded systems. Since then, timing attacks have been demonstrated on many platforms [9, 16, 24, 28, 43, 51, 69, 72, 85]. For example, Brumley et al. [24] showed timing attacks could be carried out remotely over a computer network. Cock et al. [28] found timing side channels in the seL4 microkernel and then performed a quantitative evaluation.

Noninterference properties [9, 17, 49, 56, 73] have also been formulated to characterize side-channel leaks. To quantify these leaks, Millen [64] used Shannon's channel capacity [77] to model the correlation between sensitive data and timing observations. Other approaches, including min-entropy [78] and g-leakage [10], were also developed. Backes and Köpf [14] developed an information-theoretic model for quantifying the leaked information. Köpf and Smith [58] also proposed a technique for bounding the leakage in blind cryptographic algorithms.

Prior countermeasures for timing leaks focused primarily on conditional branches, e.g., type-driven cross-copying [7]. Molnar et al. [65] introduced, along the program counter model, a method for merging branches. Köpf and Mantel [56] proposed a unification-based technique encompassing the previous two methods. Independently, Barthe et al. [17] proposed a transactional branching technique that leverages commit/abort operations. Coppens et al. [29] developed a compiler backend for removing such leaks on x86 processors. However, Mantel and Starostin [63] recently compared four of these existing techniques on Java byte-code, and showed that none was able to eliminate the leaks completely. Furthermore, these methods did not consider cache-timing leaks.

There are techniques that do not eliminate but hide timing leaks via randomization or blinding [12, 23, 31, 50, 53, 55, 88]. There are also hardware-based mitigation techniques, which fall into two categories: resource isolation and timing obfuscation. Resource isolation [61, 70, 86] may be realized by partitioning hardware to two parts (public and private) and then restrict sensitive data/operations to the private partition. However, it requires modifications of the CPU which is not always possible. Timing obfuscation [50, 74, 83] may be achieved by inserting fixed or random delays, or interfering the measurement of the system clock. In addition to being expensive, such techniques do not eliminate timing channels. Obvious RAM [42, 60, 81] is another technique for removing leakage through the data flows, but requires a substantial amount of on-chip memory and incurs significant overhead in the execution time.

Beyond timing side channels, there are countermeasure techniques for mitigating leaks through other side channels including power [54, 62] and faults [20]. Some of these techniques have been automated in compiler-like tools [8, 19, 66] whereas others have leveraged the more sophisticated, SMT solver-based, formal verification [37, 38, 89] and inductive synthesis techniques [36, 39, 84]. However, none of these compiler or formal methods based techniques was applied to cache-timing side channels.

9 CONCLUSIONS
We have presented a method for mitigating side-channel leaks via program repair. The method was implemented in SC-Eliminator, a tool for handling cryptographic libraries written in C/C++. We evaluated it on real applications and showed the method was scalable and efficient, while being effective in removing both instruction- and cache-related timing side channels. Furthermore, the mitigated software code had only moderate increases in program size and run-time overhead.

ACKNOWLEDGMENTS
This work was supported in part by the NSF under grants CNS-1617203 and CNS-1702824 and ONR under grant N00014-17-1-2896.
Eliminating Timing Side-Channel Leaks using Program Repair

REFERENCES


Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Yoshitaka Nagami, Daisuke Miyamoto, Hiroaki Hazeyama, and Youki.*


