SCASE: Automated Secret Recovery via Side-Channel-Assisted Symbolic Execution

Daniel Weber
CISPA Helmholtz Center for Information Security

Lukas Gerlach
CISPA Helmholtz Center for Information Security

Leon Trampert
CISPA Helmholtz Center for Information Security

Youheng Lü
SCHUTZWERK GmbH

Jo Van Bulck
DistriNet, KU Leuven

Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

In recent years, there has been an explosion of research on software-based side-channel attacks, which commonly require an in-depth understanding of the victim application to extract sensitive information. With evermore leakage sources and targets, an important remaining challenge is how to *automatically* reconstruct secrets from side-channel traces.

This paper proposes SCASE, a novel methodology for inferring secrets from an opaque victim binary using symbolic execution, guided by a concrete side-channel trace. Our key innovation is in utilizing the memory accesses observed in the side-channel trace to effectively prune the symbolic-execution space, thus avoiding state explosion. To demonstrate the effectiveness of our approach, we introduce Athena, a proof-of-concept framework to automatically recover secrets from Intel SGX enclaves via controlled channels. We show that Athena can automatically recover the 2048-bit secret key of an enclave running RSA within 4 minutes and the 256-bit key from an RC4 KSA implementation within 5 minutes. Furthermore, we demonstrate key recovery of OpenSSL's 256-bit AES S-Box implementation and recover the inputs to OpenSSL's binary extended Euclidean algorithm. To demonstrate the versatility of our approach beyond cryptographic applications, we further recover the input to a poker-hand evaluator. In conclusion, our findings indicate that constraining symbolic execution via side-channel traces is an effective way to automate software-based side-channel attacks without requiring an in-depth understanding of the victim application.

1 Introduction

Despite being recognized for several decades, software-based side-channel attacks remain a significant threat to the confidentiality of computations in contemporary systems, affecting a wide range of applications from web browsers [1–7] to trusted execution environments [8–23]. While side channels serve as potent leakage primitives, successfully exploiting them in practical proof-of-concept (PoC) attacks often demands substantial manual effort. Attackers are assumed to have expert knowledge of the low-level behaviors they exploit, along with a comprehensive understanding of the targeted software, including its source code vulnerabilities and binary layout [24, 25]. The latter assumption may be somewhat relaxed in certain attack strategies, such as cache templating [26], which do not always require precise knowledge of the victim's internals. However, these strategies reveal only the *presence* of secret-dependent behavior rather than disclosing the actual secret values, e.g., individual bytes of a key or password. Additionally, side channels inherently only leak *metadata* about the victim, such as memory and data access patterns, requiring further manual domain expertise to correlate such metadata patterns with the target secret.

Over the last decade, the research community has invested considerable efforts in developing practical tools to find potential side-channel vulnerabilities in applications [27–29]. Static and dynamic analysis techniques, such as statistical tests, symbolic execution, or abstract interpretation, have been shown to identify potential side-channel vulnerabilities and sometimes even pinpoint the exact location of the vulnerable code pattern [27]. However, these tools commonly suffer from false positives, and manual verification is required to confirm that a *potential* leak is an actually exploitable vulnerability. In this respect, a recent comprehensive expert survey [28] among cryptographic library developers explicitly pointed to overwhelming false-positive rates. The need for manual validation of attacks can also be seen in the context of responsible disclosure, where empirical validation of attack evidence through PoC exploits is a common requirement. Besides enabling reproduction, PoC exploits enable improved bug prioritization [30]. Thus, a key research question that received comparably little attention is: *How to automatically extract secrets from empirical side-channel observation traces?*

This paper responds to this need through an innovative combination of concrete and symbolic execution: we leverage the side-channel information gathered during an online concrete run of a victim binary to effectively prune the search space of an offline symbolic execution of the same binary. Our

resulting *side-channel-assisted symbolic execution* (SCASE) methodology enables the progressive collection of path constraints while mitigating state explosion, ultimately allowing the constraint solver to autonomously concretize the secret. Crucially, our methodology does not require any understanding of the target program or how exactly the leakage correlates with the underlying secret, solely requiring *(i)* an automated side-channel trace extraction framework to record the victim's code and data accesses at a certain granularity; and *(ii)* the annotation of the secret's location in the target binary.

Figure 1 compares SCASE to prior automated approaches, further discussed in Section 6.1. The horizontal axis distinguishes works that aim to exploit or mitigate side-channel vulnerabilities, while the vertical axis indicates the level of how general an approach is. While this work is not the first approach to automate side-channel attacks (cf. Figure 1's right-hand side), previous approaches were either restricted to a specific type of victim application target (e.g., AES [31] or secret-dependent loop conditions [32]) or a specific type of secret (media content [33, 34] or simple unstructured secrets [26]). Notably, cache-template attacks [26] are limited to binary events, such as keystroke presses, and necessitate manual effort to extract more complex secrets, like cryptographic keys. In contrast, our approach can automatically infer *non-trivial* secrets, e.g., cryptographic keys, from the *generic* target application. Furthermore, our approach is not limited to a specific type of side-channel attack.

To demonstrate SCASE in practice, we develop Athena, a PoC framework to automatically build end-to-end side-channel attacks for Intel SGX enclaves. To instantiate SCASE's requirement for automated side-channel trace extraction, Athena includes a practical profiling tool that constructs a trace of a victim enclave's code and data accesses at a 4 kB page-level granularity using controlled channels [21, 22, 35, 36]. Athena further implements a novel side-channel-assisted exploration technique as an extension to the popular `angr` [37, 38] binary symbolic execution framework, allowing to automatically recover the secrets used during enclave execution. We show that Athena can recover a 2048-bit RSA key from a Square and Multiply implementation in less than 4 minutes and demonstrate the automated extraction of a 256-bit secret key from the RC4 key-scheduling algorithm (KSA). To the best of our knowledge, our work is the first that demonstrates a single-trace key recovery attack on RC4 KSA. To showcase that Athena is capable of handling complex data-flow constraints, we recover a 256-bit AES key from OpenSSL's S-Box implementation, thus requiring Athena to automatically infer the AES master key from the memory accesses made with the round keys. As a demonstration of Athena's applicability to complex mixed control- and data-flow constraints, we recover the secret input to OpenSSL's binary extended Euclidean algorithm, which is used to generate key material for cryptographic algorithms [8, 22, 39]. We further recover the secret input to a poker-hand evaluator to showcase the versatility of our approach besides purely cryptographic applications. Additionally, we show that Athena can generate new attacks by showing a completely novel side-channel attack on RC4 key initialization.

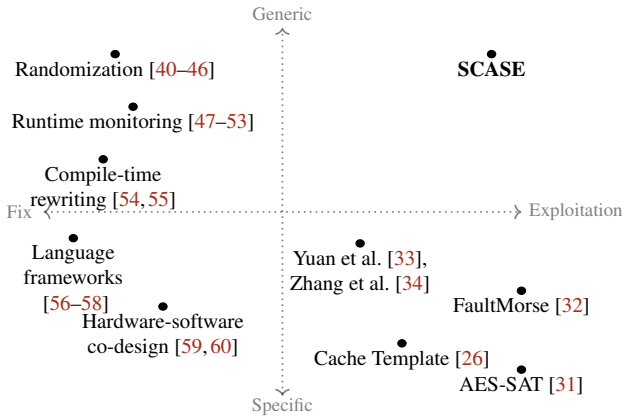


Figure 1: Categorization of related works and how SCASE fits into the landscape.

These case studies demonstrate that combining memory trace guidance with symbolic execution enables practical and efficient secret extraction without prior knowledge of victim layouts or complex reverse engineering.

Contributions. We summarize our contributions as follows.

- We present a novel approach to guide symbolic execution with memory observations from side-channel attacks. This allows for automatically recovering secrets from memory observations. Thus, we significantly reduce the manual effort required to create PoC exploits.
- We develop Athena, an open-source framework for automated secret extraction from Intel SGX enclaves, including a powerful page-access profiler and side-channel assisted exploration for the popular `angr` framework.
- We showcase Athena by recovering the private key of Intel SGX enclaves running a 2048-bit Square and Multiply RSA implementation in less than 4 minutes.
- We automatically extract a 256-bit key from an RC4 KSA, a 256-bit AES key from OpenSSL, and the input to OpenSSL's binary extended Euclidean algorithm.
- We recover a poker hand from a poker-hand evaluator, showcasing the versatility of our approach, automatically handling complex lookup structures.

Availability. Our code is available at `https://github.com/cispa/scase` and `https://doi.org/10.5281/zenodo.15609410`.

2 Background

In this section, we provide background for the paper, covering software-based side-channel attacks and symbolic execution.

2.1 Software-Based Side-Channel Attacks

Side-channel attacks refer to a class of attacks that leak meta information about a program's execution to infer confidential information about said program. The majority of software-based side-channel attacks abuse the interaction of a program with software components or the CPU's microarchitecture to leak information. The source of this leakage can be diverse, ranging from the CPU caches [61–65] over the state of internal structures of the CPU [66–68] and contention on the CPU's execution units [69–71] to its power consumption [72, 73]. Modern software-based side-channel attacks are often very powerful, achieving high accuracy and throughput [74]. Besides local scenarios, software-based side-channel attacks are also applicable in remote scenarios [75–77].

2.2 Symbolic Execution

Symbolic execution [78] is a static program analysis technique that systematically explores all reachable execution states of a program under all possible inputs. Inputs to the symbolic execution engine (SEE) can be either concrete, i.e., represented by fixed values, or symbolic, i.e., represented by symbols with associated constraints. Symbolic execution typically consists of two main phases. In the *path exploration phase*, they explore the program's control-flow graph (CFG) by emulating all possible execution paths. When encountering a branch that cannot be decided based on concrete inputs, the SEE explores both branches by forking the current execution state and continuing the emulated execution for both possible outcomes. For each branch target, the branch condition's symbolic values are then constrained to achieve the outcome leading to the current state. These constraints represent conditions the symbolic values must satisfy to reach the current state. Eventually, in the *solving phase*, a logic solver, e.g., a SAT solver, can be used in conjunction with these constraints to derive concrete values for the symbolic values that fulfill the associated logical constraints. Symbolic execution can proceed at either the source code [79] or binary levels [38, 80]. A mature example of the latter is the popular `angr` [37, 38] framework, which has found widespread use in vulnerability research on real-world binaries, including on Intel SGX enclaves [81–83]. Despite its advantages, symbolic execution is known to suffer from *state explosion*, as each fork, i.e., each branch of the program, exponentially grows the memory states and the conditions that the SAT solver of the SEE has to solve. Thus, while symbolic execution has many practical applications, e.g., for improving fuzzers [84] or detecting vulnerabilities [85–88], solving complex conditions such as cryptographically-secure encryption or hashing schemes is considered infeasible.

2.3 Intel Software Guard Extensions

Intel's Software Guard Extensions (SGX) [89, 90] is a prominent trusted execution environment (TEE) that provides runtime confidentiality and integrity guarantees for hardware-isolated memory regions, called *enclaves*. SGX enclaves live within a conventional user process and remain protected even against root adversaries that control privileged software on the target platform. SGX's ambitious isolation model and small trusted computing base make it ideal for high-security workloads, such as cryptographic libraries. SGX enclaves can be accessed through designated entry points known as `ECALL` functions, while `OCALL` functions are implemented in the untrusted host process and serve as untrusted callbacks.

Over the past decade, SGX has attracted an extensive line of offensive research [91, 92] exploiting the privileged adversary's control over the untrusted operating system to mount new [11, 20–22, 35, 93–96] or improved [9, 15, 17, 19, 36] side-channel attacks. This ongoing scrutiny has prompted Intel to denote SGX as "the most researched, updated and battle-tested TEE for data center confidential computing, with the smallest attack surface within the system" [97]. Consequently, SGX also serves as a foundational element for newer virtualization-based technologies like Intel TDX [98].

2.4 Controlled-Channel Attacks

One notable side-channel attack on TEEs, such as Intel SGX, is the controlled-channel attack [35]. In a controlled-channel attack, the attacker abuses the fact that TEE enclaves often rely on the attacker-controlled operating system to set up their memory pages. The attacker can thus manipulate the page-table entries of the enclave to fault upon access, e.g., by clearing the present bit of the page-table entry. When the enclave accesses a faulting memory page, the enclave execution is halted via an asynchronous exit of the enclave, and control is transferred to the OS. While modern enclaves clear registers during asynchronous exit, the information that the specific page-table entry was accessed, and hence a memory access to the underlying page occurred, is leaked.

Another notable attack technique on TEEs is single-stepping [36]. Single-stepping works by programming a timer interrupt to occur precisely after executing exactly one instruction of the enclave code. Van Bulck et al. [36] showed that single-stepping gives the attacker precise control over the execution of an Intel SGX enclave. Due to the fine-grained execution capabilities that single-stepping gives an attacker, it also improves on the temporal resolution of controlled-channel attacks. Single-stepping is not restricted to SGX but can be applied to other TEEs and secure VMs, e.g., Intel TDX [99] or AMD SEV [100, 101]. To ease attacks on SGX, Van Bulck et al. developed the SGX-Step framework [36]. Besides functions to modify page-table entries, the framework also provides functions to single-step an SGX enclave.

3 SCASE

We present SCASE (Side-Channel-Assisted Symbolic Execution), our novel approach to automatically recover secrets from a victim program. On a high level, our approach uses information obtained from a side-channel attack to guide an SEE. This guidance empowers the SEE to overcome previous limitations and automatically recover the victim's secrets.

3.1 General Threat Model

In our threat model, we assume that the attacker has access to the target binary. Note that this enables the attacker to execute the victim symbolically. The attacker aims to recover a secret, e.g., a private key, from the target application. We assume the attacker cannot directly extract the secret and can only leak meta information about the victim application, i.e., observe a side channel. Besides this meta information leakage, we assume no further vulnerabilities in either hardware or software. Furthermore, we assume that the attacker has all the capabilities required to monitor this information, e.g., many side-channel attacks require native code execution [61–64].

3.2 Overview

To ease the generation of PoC exploits for secret leakage, we propose SCASE, a combination of side-channel attacks and symbolic execution. SCASE uses memory traces obtained from a side-channel attack to guide the path exploration and constraint creation of an SEE. With the help of the side-channel traces, the SEE can recover the value of the symbolized target secret, as the search space is drastically reduced. Figure 2 shows an overview of our approach, consisting of an *online* and an *offline* phase. In the *online phase*, a side-channel attack is mounted on the victim application that tries to infer as much information as possible about the victim application's memory access patterns. We refer to the information recovered by this side-channel attack as *memory trace*. The content of the memory trace is split into control-flow- and data-flow-related memory accesses, which we refer to as *control-flow trace* and *data-flow trace*, respectively.

In the *offline phase*, these traces support an SEE to recover the victim's secrets. Figure 3 illustrates how the offline phase leverages the traces from the online phase. The *control-flow trace* conveys information about the control-flow decisions of the victim, i.e., which branches were taken. The control-flow trace can be used to prune execution the victim did not take during the online phase. Figure 3a illustrates this concept by displaying that side-channel information (cornered nodes) can be used to prune away states not reached during the online phase. This pruning significantly reduces the state space that the SEE has to explore. Furthermore, secret-dependent control flow, e.g., `if`-branches based on the secret value, leads to constraints on the victim's secret encoded in this trace.

The *data-flow trace* conveys information about the data flow of the victim, i.e., which memory locations were accessed. Thus, encoding secret-dependent memory accesses, e.g., array lookups based on the secret value. Hence, knowing the data-flow access patterns allows the SEE to prune states where the data-flow access pattern does not match the observed one. Figure 3b displays this concept by illustrating that a memory access containing information about the memory access f to node F can be used to prune away state G. Despite conveying information that allows further pruning of states, data-flow memory traces can encode additional constraints on the secret value.

To summarize, after mapping the entries of both memory traces to the state space of the SEE, we can prune the state space and infer additional constraints on the secret. Hence, in the offline phase, the SEE can efficiently reason about the secret of the victim due to the meta information obtained in the online phase.

3.3 Introductory Toy Example

Figure 4 illustrates the general idea of our approach. The code implements a simple cryptographic algorithm that uses a secret value as an index to a lookup table and enters a loop that squares the variable `ct` whenever the current bit of the secret key, i.e., the variable `SK`, is 1. Despite the simplicity of the code patterns, they are widely used primitives, e.g., lookup tables in AES [102] and multiplication algorithms used for RSA computations [103].

Executing the code with the secret key `SK = 0b1101` yields a memory trace, as shown in the middle of Figure 4. This memory trace is obtained in the online phase and illustrates the meta information leaked by a side-channel attack. In this case, the side-channel attack leaks the branch decisions of Line 10 to 12 as well as the fact that a memory access was made to the lookup table, i.e., the variable `lut`. While such a memory trace may contain enough information to recover the secret key directly, recovery requires a precise understanding of the initial algorithm. In this case, a human expert needs to understand that branching in line 12 corresponds to a '1'-bit in the secret, while a not branching refers to a '0'-bit.

In the offline phase, we symbolize the secret, i.e., `SK`, and let an SEE find the concrete value. While SEEs are designed to recover input for generic programs, this does not work directly for complex programs such as cryptographic algorithms. SEEs reason over all possible states of a program, i.e., the CFG (see right of Figure 4). Thus, the SEE iterates over all paths, leading to a doubling of the state space for every encountered branch. Hence, the state space that has to be explored quickly becomes infeasibly large Section 2. In our toy example, the number of states, i.e., the state space, is already 2^n for n rounds.

For SCASE, we combine the online and offline phases. While the memory trace, recovered by a side-channel attack,

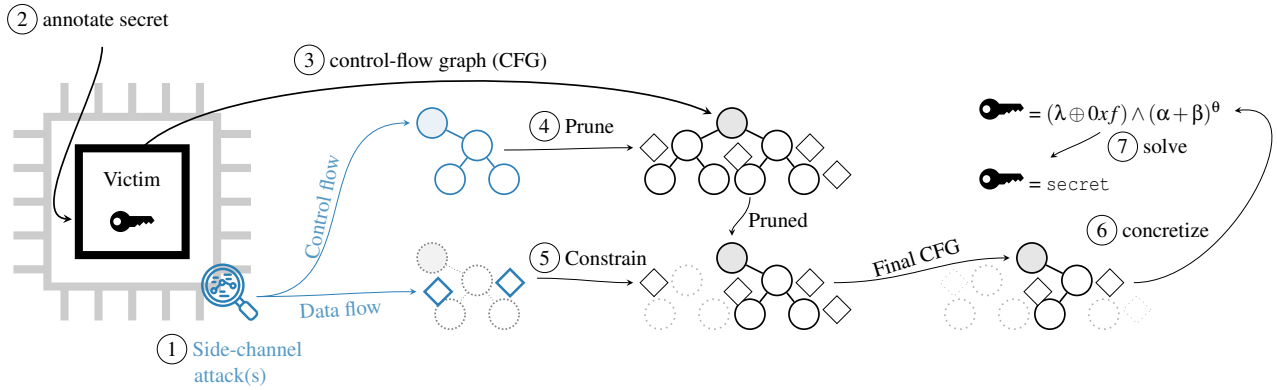
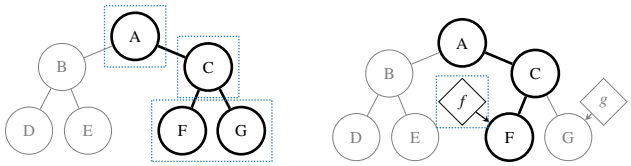


Figure 2: In the online phase, SCASE mounts a side-channel attack on the victim (①) to obtain data- and control-flow traces. In the offline phase,SCASE annotates the secret (②) and creates a CFG (③) of the victim. The previously obtained traces are then used to prune states from the CFG directly (④) and indirectly via constraints (⑤). The final reduced CFG allows symbolic execution to concretize (⑥) the symbolized secret without state explosion, resulting in the automated extraction of the actual secret (⑦) used by the victim.



(a) Side-channel information on the *control* flow (nodes A to F) helps the SEE to prune states (B,D,E) not reached during execution.

(b) Side-channel information on the *data* flow (nodes f and g) helps prune states (G) where the access pattern does not match the observed one.

Figure 3: The side-channel traces from the online phase (blue dotted rectangles) enable the symbolic execution in the offline phase to prune states to a manageable number of states for which the SEE can find concrete values.

does not directly contain the key, it provides information on which branch was taken. Thus, the execution trace removes the need for the SEE to explore every possible path, leading to a significant reduction of the state space that has to be explored. In other words, for every branch contained in the memory trace, the state space for the SEE no longer doubles, as we can guide the SEE to the chosen branch target.

3.4 Major Challenges

To achieve the goal of fully automating the secret extraction, we have to overcome various challenges.

C1: State Explosion. The feasibility of symbolic execution is limited by the state explosion problem, i.e., the exponentially growing state space required to model all possible execution paths of a program (cf. Section 3.3). For every branch, the outcome of the branch decision is not concrete, the SEE has to fork the current state. While this is a key aspect of SEEs, as it allows to reason about all theoretically possible states, it is also the main reason why symbolic execution is infeasible for complex programs. The state explosion is a particular challenge for cryptographic algorithms as their security guarantees are rooted in having a large state space. Hence, secret-dependent control-flow of a typical cryptographic application leads to an exponentially growing state space. Another problem using symbolic execution to recover complex secrets is that the SEE has to constrain the secret value. For a typical, i.e., unguided, SEE, this requires additional information, e.g., the ciphertext that was created by cryptographic algorithm. Otherwise, the SEE could only output all possible final states, even when assuming that the state explosion would have been solved. As both problems are rooted in the growing number of states, we consider both problems as a major challenge. For SCASE, we tackle this challenge by using the memory access information obtained from the online phase, i.e., a side-channel attack, to reduce the number of branch targets that the SEE has to explore.

C2: Map Side-Channel Information to the SEE's State Space. To make use of the side-channel information in the SEE, we have to map these real-world observations to the emulated execution of the SEE. This step is crucial as for each memory access that the SEE encounters in *any* state, we need to know to which memory access in the memory trace it corresponds. In other words, we need to establish an injective mapping between memory trace entries and the emulated memory accesses for *potential* states of the SEE. The mapping is injective as for every memory trace entry, there is *exactly one* corresponding execution state in the SEE. Note that each virtual address, e.g., each assembly instruction or data in the target application, can be accessed multiple

```
1  // secret key
2  nibble SK = <...>;
3  // plaintext
4  nibble PT = <...>;
5  size_t ROUNDS = 2;
6
7  nibble ct =
8  lut[(SK >> 2) ^ PT];
9
10 for (int i = 0;
11   i < ROUNDS; i++){
12  if (SK & 1)
13    ct = ct * ct;
14   SK >>= 1;
15 }
```

```
memAccess(
  lut[0b11^pt])
br_line11(true)
br_line12(false)
br_line11(true)
br_line12(true)
```

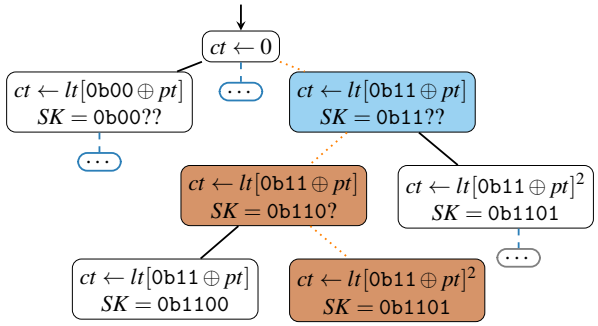


Figure 4: Code snippet of a toy cryptographic algorithm (left), a corresponding memory trace (middle), and its CFG (right). The side-channel trace encodes the branching decisions of the algorithm, which are highlighted in the CFG.

times during the execution. Hence, a correct mapping needs to ensure that each memory trace entry is mapped to the correct execution state of the SEE, i.e., the correct and exact point in time of the emulated execution. Note that a correct mapping is especially challenging in the presence of a noisy side-channel attack as a single incorrect mapping can lead to incorrect conclusions, thus potentially preventing the approach from recovering the target application's secret. Hence, it is crucial to establish a noise-free and correct mapping. Depending on the side-channel attack and the attacker's capabilities, there exist different ways to tackle this challenge (cf. Section 3.5).

C3: Separating Control- and Data-Flow-Related Memory Accesses. While both, control- and data-flow-related memory accesses, can be used to guide SEEs, they impact the emulated execution in different ways. To precisely guide the emulated execution, it is not sufficient to know that *any* memory access has taken place at a given state of the emulation. Instead, we need to know whether the memory access stems from the fetching of an instruction from memory or the execution of the current instruction. Being able to distinguish these two cases allows to reason about misalignments between the real-world trace and the emulated execution. Note that this separation is not a strict requirement for our approach but significantly improves the guidance of the SEE by benefiting both the pruning of state spaces and the constraint creation.

3.5 SCASE: Recovering Secrets from Memory Traces using Symbolic Execution

In this section, we describe the details of SCASE and how we handle the previously introduced challenges. We discuss how memory traces can be split into control- and data-flow traces, thus tackling **C3**. We explain techniques of creating an injective mapping between the memory traces and the states of the SEE, thus tackling **C2**. We elaborate on how overcoming **C2** and **C3** allows us to use the memory trace's information to prune the SSE's state space while adding further constraints on the secret value, thus tackling **C1**.

Splitting Memory Traces into Control- and Data-Flow Memory Traces. For precise guidance, the attacker has to separate control- and data-flow-related memory accesses (cf. **C3**). The reason is that for optimal guidance of the SEE, it is beneficial to know whether a memory access at a certain position was caused by the CPU frontend fetching the instructions or by the actual execution of the instruction. Hereby, the former means that the address of the current instruction has to match the next entry in the memory trace. The latter means that the memory trace entry instead has to match the semantics of a memory-accessing instruction. The potential methods of separating control- and data-flow-related memory accesses depend entirely on the side-channel attacker's capabilities. In a strong attacker model, where the attacker has operating system capabilities, e.g., in the threat model of trusted-execution environments, the attacker can read the permission bits of the victim's memory pages. The permission bits are control bits of the page-table entries associated with each memory page. Along other aspects, these bits control whether a page is writable or executable, whereas the latter is a requirement to execute code stored in a memory page. On modern CPUs, attackers can rely on the bit in the page table entry indicating whether a page is executable. Our PoC implementation Athena (cf. Section 4) uses this type of separation and shows that it is a feasible method to separate control- and data-flow-related memory accesses for various target applications. Alternatively, a less privileged attacker can reason whether a memory access stems from the CPU frontend by observing the state of microarchitectural components. For example, the instruction cache or the iTLB are only affected by frontend memory fetches. Thus, monitoring them allows to reason about the type of memory accesses.

Creating a Mapping between Memory Trace Entries and Emulated States. Creating an injective mapping between

the memory trace entries and each potential state of the SEE is required to effectively guide the SEE (cf. **C2**). Creating such a mapping is easiest when the attacker ensures that the memory traces are error-free. Here, methods to ensure error-free memory traces again depend largely on the side-channel attack. To achieve this, the attacker can use multiple runs of a side-channel attack to cross-verify the information contained in the resulting memory traces. Alternatively, side-channel attacks exist that do not contain errors in the first place, e.g., controlled-channel attacks [35]. Once the memory traces are free of errors, the attacker needs to map the entries of the memory traces to the SEE's state space. For this, the most intuitive and straightforward approach is to map memory trace entries based on their virtual memory addresses. The attacker has to ensure that each memory trace entry is mapped to the exact point of access in the emulated execution. This is because a given address can be accessed multiple times during the execution of a program. Hence, the implementation of SCASE has to maintain the current position for both the control- and data-flow memory trace for any exploration that is emulated by the SEE. This means that every time the SEE has to fork, another position in the memory trace must also be maintained. A potential implementation of this is to increase an index for each emulated memory access to the corresponding memory trace file and to maintain a different index for every fork of the SEE's states. Note that while virtual memory addresses are not the only feasible mapping between side-channel information and the emulated execution, they are applicable for a range of side-channel attacks, e.g., Flush+Reload [61], Flush+Flush [62], the `umwait`-based TLT side channel [104], and controlled-channel attacks [35]. We discuss alternative injective mappings in Section 4.4.

Pruning the State Space and Adding Additional Constraints. The traces, which are mapped to the SEE's representation, are used to prune states that are not reached during the actual execution and to further create constraints on the secret (cf. **C1**). For example, when exploring the first level of the CFG of Figure 4, we adapt the SEE to ignore the first two child nodes as the control-flow memory trace yields the information that the third node was executed when the victim application computed on its secret. In the example of Figure 4, knowing that the branch in Line 10-11 was true when the branch is first encountered, adds the constraint `i < ROUNDS` for the given node in the CFG. Note that the control-flow memory trace does not always contain information about the path taken for every branch decision.

Besides pruning the state space, constraints stemming from the memory traces are added to the SEE. For example, when the exploration hits the third node of the CFG in Figure 4, we add the constraint that at this the execution path, the memory access to the lookup table, i.e., `lut`, was made at index `0b11 XOR pt`. The resulting constraint, i.e., `addr(LUT) + (SK » 2 XOR PT) == ObservedAddress`, allows the SEE to further reason about the secret input.

After finishing the path exploration phase, the SEE can use its solver to recover the confidential information, e.g., the private key, by concretizing the constraints added to the secret. Depending on the complexity of the victim binary and the exact memory accesses that were monitored, solving for the secret is feasible. Generally speaking, all types of attacks that purely rely on the memory access behavior of an application can be automated using this approach. The reason is that these memory accesses are encoded in the memory traces and thus be converted to constraints for the solver of the SEE.

4 Athena Framework

In this section, we discuss the implementation of our PoC framework Athena. Athena implements SCASE with controlled-channel attacks as a side channel. Thus, Athena automatically generates exploits for side-channel vulnerable Intel SGX enclaves. While SCASE is not limited to Intel SGX or trusted-execution environments in general, controlled-channel attacks provide a powerful way of automating the online phase to obtain noise-free traces. For Athena, we implement 2 main parts: an automated controlled-channel attack for Intel SGX based on single stepping and monitoring access bits as the online phase (Section 4.2), and an exploration strategy for the SEE angr [37, 38] as the offline phase (Section 4.1).

Concrete Threat Model. The threat model for Athena extends and concretizes the threat model of SCASE (cf. Section 3.1). In line with the Intel SGX threat model, which is also commonly used in related work [24], we assume a privileged native-code attacker, which includes the ability to load arbitrary kernel modules, e.g., the SGX-step module [36]. Note that as we assume full knowledge over the content of the enclave beside the target data, the attacker can execute a copy of the victim application, e.g., as a debug enclave [89] or in an emulator, without the actual secrets.

4.1 Offline Phase

Emulating the Target. In the offline phase, Athena utilizes Guardian [83] to make angr [37, 38] compatible with Intel SGX enclave binaries. This is possible since Intel SGX enclaves are represented by ELF files, which allows to execute specific enclave functions without the need for conversion. Additionally, the code of enclaves is attacker-accessible in the SGX threat model [89]. To enable executing enclave functions that use `OCALL`s, i.e., functionality for the enclave provided by the host application, `OCALL`s can be hooked by Guardian to return dummy values. For applications that do not work with dummy values, angr's `SymProcedures` feature can be used to create custom hooks. Since the result of `OCALL`s generally cannot be trusted as it represents the result of an untrusted computation, a situation where the result of an `OCALL` is used

to derive a secret or as part of a confidential computation is unlikely. Note that emulating the enclave binary does not allow extracting secrets, as any secret is typically loaded from a secure remote instance only after successful remote attestation. Even an attacker fully aware of the target enclave's code and data cannot impersonate a remote attestation successfully.

Mapping the Leakage to angr's State. As control- and data-flow-related memory accesses contain information that has to be handled differently for an optimal recovery, we split the memory trace into two parts. Control-flow-related memory accesses, i.e., accesses to executable memory pages, convey information about which branch decisions were taken by the victim program. Hence, we use this information to guide the exploration phase based on the concept described in Section 3. More specifically, we create a new `ExplorationTechnique` subclass for angr, allowing us to precisely control how `angr` symbolically executes binaries. A crucial part is the `step` member function of the `ExplorationTechnique` class. The `step` function is called every time any possible path that `angr` explores encounters a control-flow branch, i.e., every time a new basic block is reached. Single-stepping the victim enclave during memory trace creation yields exactly one memory access per instruction pointer increment. Thus, we keep track of the expected memory access, i.e., the position in the control-flow trace, by incrementing the index into the control-flow trace based on the number of instructions in the current basic block. This allows us to create a `step` function that checks on every branch decision whether the memory access pattern differs from the control-flow memory trace. If so, we drop the execution state and restrict the exploration to paths that align with the memory trace. As discussed in Section 3, this effectively prevents the state-explosion problem that would occur otherwise. It is important to note that by dropping branch targets that do not align with the memory trace, we implicitly encode the constraints given by the side-channel leakage into the SAT-solving engine of `angr`.

The data-flow memory trace is handled differently as it does not correspond to changes in the instruction pointer but instead adds explicit constraints to the memory accesses made by the program, i.e., that at a specific point of the execution a specific memory access was made. To model this, Athena registers callbacks on all memory reads and writes made by `angr`. Similar to the control-flow traces, the code in these callbacks keeps track of the current position in the data-flow trace for a given execution state. Note that calculating the position in the data-flow trace is trivial, as per definition, every invocation of the callback just increments the position by 1 for its respective execution state. Thus, all that is left is actually constraining the solver to a memory access that aligns with the data-flow trace. For this, we directly add the constraint `mActual AND` $\texttt{NEG}_{\texttt{64-bit}}$`(0xFFF) == mTrace` to the SAT solver, where `mActual` denotes to the actual memory address accesses at the specific point of the execution, `mTrace` corresponds to the page-aligned entry in the data-flow-trace, and $\texttt{NEG}_{\texttt{64-bit}}$ is the binary negation of a 64-bit integer. The previous formula assumes a page size of 4 kB and that we only add the constraint for a memory access at one *specific* point of the execution state, i.e., a memory load inside a loop can have different constraints for each iteration of the loop despite having the same memory address in every iteration.

Symbolizing and Constraining the Secret. For recovering the secret, Athena requires the user to symbolize the memory range holding the secret bytes (cf. Section 4.3). Athena then adds data- and control-flow constraints, stemming from the side-channel memory-access trace, to the symbolized secret when exploring the victim program's control-flow graph. Note that this allows our approach to also recover partial secrets, even when the side-channel trace does not provide enough information to fully disclose the secret (cf. Section 5.3).

In the broader landscape of symbolic-execution techniques [105], our approach represents a novel form of concolic or dynamic symbolic execution, as it integrates both symbolic exploration and concrete execution traces. While Athena conducts a *symbolic* exploration of the target binary, it uses the side-channel traces from a *concrete* execution to prune states, whenever possible, and optimize the exploration.

4.2 Online Phase

Athena requires memory traces that are as fine-grained as possible. For our PoC implementation, we mainly use controlled-channel attacks to create these memory traces, as they provide fine-grained information and were used on various target applications in previous work [18, 35, 36, 55]. We mount a controlled-channel attack to observe the memory access patterns of the victim enclave. More precisely, we follow the approach of Moghimi et al. [22] and monitor the access bits of the page-table entries while single-stepping through the victim enclave. We divide the page-table entries into two sets based on the entry's "NX" bit and clear all "accessed" bits. If the page is marked as non-executable, we assume it only holds data and track it as part of the *data-flow trace*. If the page is not marked as non-executable, we assume it holds code, and hence we track it as part of the *control-flow trace*. This differentiation is beneficial when the memory traces are handled by `angr`, as it optimizes the information gain. Note that splitting the traces can also be done by an unprivileged attacker who knows the memory map of the victim application. When knowledge of the memory map is not available, the victim binary itself gives this information for all statically defined memory pages. Alternatively, previous work used side channels to determine whether a page is executable [106].

We exclude memory pages from the traces that technically belong to the victim but do not give meaningful information about the access pattern of the victim, such as the memory pages of the enclave thread control structures (TCS). Note

that the attacker application starting the SGX enclave is given the memory location and size of the TCS and other excluded memory pages, which reduces this step to a mere parsing of attacker-available information. Eventually, we use SGX-Step [36] to single-step through the enclave while observing the accessed bits of the page-table entries. If an accessed bit is set, we append it to the corresponding trace, i.e., control- or data-flow trace, and clear it again before resuming the execution of the victim enclave. The result are two traces, i.e., ordered lists of memory pages, that correspond to the control- and data-flow access pattern of the victim enclave.

4.3 Usage

Athena allows to automatically retrieve the secret from a victim enclave by only adapting as few as 3 lines of code for a specific target application. The implementation is almost entirely automated, requiring only 2 user interactions. For the memory tracing to work, the tracer needs to call the vulnerable `ECALL` of the target enclave. For this, the user needs to implement a callback that executes the `ECALL`. During our experiments, these callbacks were 1 to 3 lines of C code, as they are essentially just a function call with appropriate arguments. Note that for simple `ECALL`s, this step can be further automated by parsing the type information of the `ECALL` arguments and automatically creating a call with appropriate arguments, similar to the approach by Cloosters et al. [82].

The offline phase of Athena, i.e., the actual extraction of the secret from the memory traces, is executed by calling a corresponding Athena Python module. This only requires the user to annotate the secret that should be recovered. Section A shows an example of how the Athena Python module is used. Section A matches a secret recovery from the function `encrypt(const char* key, const char* key_len, const char* msg)`. As dictated by the System-V AMD64 ABI, the first argument, i.e., the secret key, is passed in the `RDI` register, while the second argument, i.e., the key length, is encoded in the `RSI` register. Thus, lines 15-17 instruct the framework to retrieve the secret key passed in the `RDI` register. Note that only the highlighted 3 lines and the parameters specifying the target `ECALL` and target function need to be adapted when the target changes. Due to the combination of side-channel information that the framework automatically obtains and encodes in an SEE, Athena reduces the effort of creating a PoC for a vulnerable enclave to a bare minimum. Hence, Athena makes PoC generation faster for experts and accessible to a broader audience.

4.4 Alternative Side-Channel Traces

While our PoC framework uses a controlled-channel attack to monitor the victim program, many other software-based side-channel attacks would work similarly.

Spatial Resolution. One major difference between controlled-channel attacks and most other software-based side-channel attacks is that most of them cannot monitor every victim page. For example, an attacker mounting Flush+Reload needs to restrict their attack to a much smaller set of addresses that can be monitored, as thousands of pages cannot be monitored reliably in parallel using Flush+Reload. While this is a drawback compared to controlled-channel attacks, most other software-based side-channel attacks have a finer granularity than controlled-channel attacks, e.g., cache-line granularity, which gives the attacker more precise information for recovering secrets. Athena can easily be adapted to work with side-channel attacks that do not monitor *every* memory page. As an attacker knows the set of monitored memory pages, this information can be used in Athena to restrict the guidance of all memory accesses to the set of monitored pages.

Athena could be further extended to support side-channel attacks that do not work directly on virtual addresses but instead on microarchitectural structures, e.g., cache sets for Prime+Probe. For this, one has to simulate the cache set behavior in the SEE, i.e., the initial state of the monitored cache set and how it changes over time. While this is more challenging than modeling attacks where a mapping based solely on virtual memory addresses can be used, implementing this is merely an engineering task.

Dealing with Noise. Another considerable difference between controlled-channel attacks and most other software-based side-channel attacks is that the latter are often not noise-free, e.g., cache attacks may be impacted by false positives and false negatives. Importantly, previous research [16, 17, 74] has convincingly demonstrated that reliable information can be extracted by repeating an attack multiple times. To empirically validate this, we implement a Flush+Reload attack on the square-and-multiply algorithm implemented in mbedTLS 2.16.1. When decrypting a ciphertext with a 1024-bit key, we observe 68 out of 100 traces to be entirely error-free. Thus, by running the attack 3 times and using majority vote, we can straightforwardly construct a noise-free trace without errors.

While correcting all errors for Prime+Probe is harder than for Flush+Reload, previous research [16, 17] has shown that the error rate of a Prime+Probe attack can be overcome by repeating an attack multiple times. To empirically validate this, we implement a Prime+Probe PoC on a program with a secret-dependent branch processing 2048 secret bits. To remove the errors, we repeat the complete attack multiple times, while using a majority vote over the resulting leakage pattern until we reach a stable leakage pattern for 30 consecutive runs. After, on average, 115 repetitions, we reach a stable leakage pattern, which correctly encodes the 2048 bit secret key in 98 % of our tests. Thus, while it is more challenging to gather stable traces from more noisy side-channel attacks, it

is feasible to do so by repeating the attack and pre-processing the traces before feeding them into Athena.

5 Evaluation

In this section, we evaluate SCASE, focussing on our PoC implementation Athena. First, we evaluate how well the memory-trace-guided symbolic execution performs using artificial traces (Section 5.1). Second, we analyze the performance of Athena with real-world SGX traces, similar to Moghimi et al. [22] (Section 5.2). Afterward, we evaluate Athena on four further realistic examples.

5.1 Efficiency of Secret Recovery

In this section, we evaluate how different granularities of memory traces improve the path exploration of angr [37, 38], the symbolic execution framework used in Athena. We leverage a custom victim program that exhibits control- and data-flow leakage via a jump table with secret-dependent indices. This setup allows for measuring the influence of the memory traces on the SEE in a controlled and precise manner. To evaluate the influence on a cryptographic target, we implement the square-and-multiply algorithm as a victim program. Our evaluation leverages a custom angr-based memory-tracing tool that allows us to generate memory traces of the victim program with varying granularities. We use an Intel Core i7-9700K CPU with 32 GB memory for the evaluation.

5.1.1 Jump-Table Example

The custom victim program features a secret input that consists of n-bytes, each from the range 0 to 15, i.e., a hexadecimal digit. The program converts the secret into a numerical value via value-specific parsing functions and lookup tables by interpreting each byte as a hexadecimal digit. Hereby, the program iterates over the secret and uses each byte of the secret as an index into a jump table. The jump table contains 16 entries, each pointing to a different function that writes the corresponding numerical value of the byte to a buffer. The program therefore exhibits both control and data flow leakage of the secret during the conversion of the secret.

First, we examine the ability of Athena to recover the secret for varying granularities of the memory traces. We rely on byte-granular memory traces of the victim program using our custom memory-tracing tool. We vary the granularity of the memory traces by zeroing out the lower bits of the memory addresses contained in both traces. Thus, we achieve byte granularity by zeroing out no bits and page granularity by zeroing out the 12 least significant bits. Figure 5 shows the runtime of Athena for recovering a 64-byte secret with memory traces of different granularities. Each data point is the average of 10 runs, each with a different pseudorandom secret. The left side of the x-axis represents byte granularity,

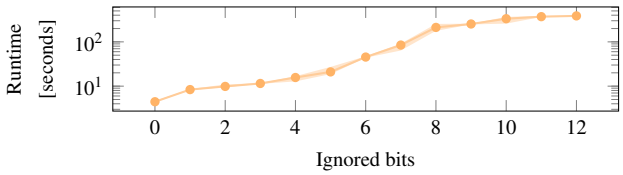


Figure 5: Athena's runtime for recovering a 64-byte secret with varying memory trace granularities. Zeroed-out bits of the memory addresses in the control- and data-flow traces are on the x-axis. The error band shows the min and max runtime.

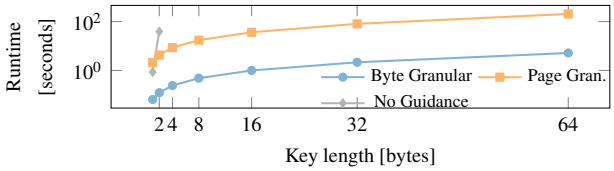


Figure 6: Athena's runtime for recovering varying-length secrets, from 1 to 64 bytes.

while the right side represents page granularity. Note the logarithmic scale of the y-axis. We observe that the runtime increases exponentially with the decreasing granularity of the memory traces. Further, we notice that the runtime of Athena is almost entirely spent in the path exploration phase, while the runtime of the solving phase is negligible. Intuitively, the runtime grows exponentially because the more information the memory traces contain, the more effective the SEE can prune away states during the path exploration phase, i.e., the overwhelming majority of the runtime.

Second, we evaluate the performance of Athena for recovering the secret with varying secret lengths. This evaluation also relies on byte-granular traces from our custom angr-based memory-tracing tool. We evaluate byte and page granularity, i.e., the most and least granular memory traces. In addition, we compare the performance of Athena against the regular performance of `angr` without any memory-trace guidance. Figure 6 shows the runtime of Athena for recovering the secret with varying secret lengths. The x-axis shows the number of bytes of the secret, while the y-axis shows the runtime in seconds. Note the logarithmic scale of the y-axis. We observe that the runtime of Athena increases *linearly* with the secret length. Furthermore, we observe that the runtime of Athena is significantly lower than the runtime of `angr` without memory-trace guidance, which fails to recover the secret for secret lengths above 2 bytes in under 10^3 seconds. This is due to the exponential state explosion that `angr` encounters during the path exploration phase without memory-trace guidance. The evaluation shows that memory-trace guidance allows efficient secret recovery, even for long secrets.

```
1  uint64_t mod_exp(uint64_t base,
2      const char* exp, size_t exp_len) {
3    uint64_t result = 1;
4    for (int i = 0; i < exp_len; i++) {
5      result = square(result);
6      if (exp[i] == '1')  // leakage
7        result = multiply(result, base);
8      result = result % N;
9    }
10   return result;
11 }
```

Listing 1: Square-and-multiply victim used during evaluation. Assuming that the variable `exp` is secret, the function does a secret-depending memory access in line 7. Thus, the memory access pattern of the program leaks information about `exp`.

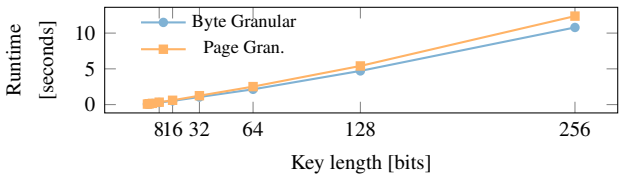


Figure 7: Athena's runtime for recovering the secret in the Square and Multiply example with varying secret lengths.

5.1.2 Square and Multiply Example

Listing 1 shows the square-and-multiply implementation calculating on a secret exponent, i.e., the variable `exp`. In line 7, the function performs a secret-dependent branch, leading to the conditional execution of `multiply`. For a sufficiently large exponent, retrieving the exponent is infeasible for an SEE. During our tests, 16 exponent bits already exceeded a runtime of 10 minutes for `angr` without memory-trace guidance. The code of the functions `square` and `multiply` resides on different pages.

We evaluate Athena's performance with byte- and page-granular traces from our memory-tracing tool. We choose secret exponents 2^n, whereas n ranges from 1 to 8. For each exponent, we execute 10 runs, each with a different pseudo-random secret. Figure 7 shows the runtime of Athena for recovering the secret with varying lengths. The x-axis shows the secret length, while the y-axis shows the runtime in seconds. We observe that the runtime of Athena increases *linearly* with the secret length. Note that the y-axis is not scaled logarithmically. The runtime of the solving phase is negligible compared to the runtime of the path exploration phase.

5.2 SGX Traces with Athena

In this section, we evaluate Athena on traces recovered from an SGX enclave using single-stepping. This experiment shows that Athena can be used on memory traces obtained from a controlled-channel attack. We recover the secret of the previously discussed square-and-multiply example (cf. Listing 1) running inside an SGX enclave, reachable via an `ECALL`. To model a realistic scenario, we use a 2048-bit key for the exponent, a typical key length for RSA.

We execute the SGX enclave and recover the memory traces on an Intel Core i3-7100T CPU with 8 GB DRAM. As recovering a 2048-bit key is memory-intensive, we execute the offline phase of Athena on an Intel Core i9-12900K CPU with 104 GB DRAM. Note that 104 GB DRAM is still reasonable for an attacker. In line with the previous experiments, we randomly generate secret keys while timing the exploration and solving phase of Athena. We recover the traces using the memory tracing approach discussed in Section 4.2, similar to Moghimi et al. [22]. Our evaluation shows that Athena can successfully recover all 2048 bits of the secret in 10 out of 10 cases, i.e., a 100 % success rate. Furthermore, the average runtime for the guided exploration phase is 195.7 s with the minimal and maximal observed runtime being 194.3 s and 196.8 s, respectively. For every execution, the time spent in the solving phase was below 10 ms. The DRAM usage of Athena spiked at 62 GB. We conclude that Athena can successfully recover complex secrets from SGX enclaves using memory traces obtained from controlled-channel attacks.

5.3 Recovering AES S-Box Keys

To demonstrate Athena's ability to leverage complex data-flow constraints, we target OpenSSL's AES S-Box implementation. Athena successfully reconstructs the AES master key from the S-Box lookups used in the substitute step.

Overview. A common optimization in software-based AES implementations is the use of lookup tables containing pre-computed values of either complete AES rounds [107] or single steps of the AES algorithm [108]. For example, the substitution step of an AES round can be sped up by looking up the secrets in a precomputed lookup table, i.e., the AES S-Box. As the values looked up in the S-Box depend on the round keys, which are derived from the master key, these accesses leak information about the master key. As the S-Box is only 256 bytes large, an attacker requires fine-grained side-channel information to distinguish between its different entries. For our evaluation, we target the AES master key passed to OpenSSL 1.0.2p's `AES_encrypt` function and let Athena automatically reconstruct the master key from the S-Box lookups. Note that for 256-bit AES keys, the derivation of the master key from the round keys is more complex than for smaller key sizes, as it must be reconstructed from the combination of multiple round keys.

Results. Table 1 summarizes Athena's runtime and key bit recovery for varying side-channel trace leakage granularities. The results stay the same, when restricting Athena to just data-flow leakage. When restricting Athena to pure control-flow leakage, it fails to recover the master key since the control

Table 1: Athena's performance for recovering 256-bit AES keys with varying leakage granularities.

Granularity (Bytes)	Key Bits Recovered	Runtime
1	256 bits (100.0 %)	2.68 s
2	182 bits (71.1 %)	8.40 s
4	178 bits (69.5 %)	30.95 s
8	174 bits (68.0 %)	93.11 s
16	161 bits (62.9 %)	461.37 s

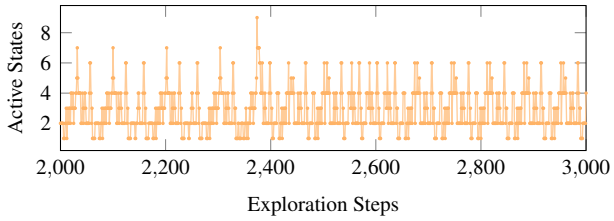
flow does not depend on any secrets. Thus, we observe that byte-granular data-flow leakage from the S-Box yields enough information to recover the entire master key. More coarse-grained leakage only allows partial key recovery. As expected, the runtime grows exponentially with reduced information.

An important aspect of the partial key leakage stemming from our approach is that, in contrast to partial key leakage due to noise, the recovered constraint makes it clear which of the resulting key bits are fully constrained, i.e., reported as recovered, and which are unconstrained. Thus, post processing techniques, e.g., bruteforcing of the remaining bits, can be restricted to the unconstrained bits. From this, we conclude that when armed with a side-channel attack that can differentiate between all entries of the S-box, e.g., the prefetcher side channel introduced by Hetterich et al. [109], Athena can automatically reconstruct the master key from the meta information gained about the different round keys. This case study demonstrates the ability of Athena to recover secrets from complex data-flow constraints.

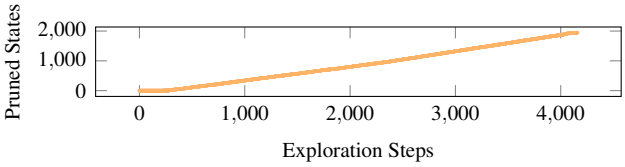
5.4 Exploiting OpenSSL's Binary Extended Euclidean Algorithm

While the previous experiments demonstrate that Athena can handle both control- and data-flow constraints, this case study demonstrates Athena's handling of *combined complex* data- and control-flow constraints. For this, we target OpenSSL's implementation of the binary extended Euclidean algorithm (BEEA). Most notably, the BEEA is used during the key generation process of cryptographic algorithms, such as RSA, DSA, and ECDSA, and is a popular target of prior manual attacks [8, 22, 39].

Overview. The BEEA, an optimized variant of the extended Euclidian algorithm, is used to compute the greatest common divisor and the modular inverse of two numbers. We target the BEEA implementation in OpenSSL 1.1.0h, as displayed in Section B. The implementation does not directly iterate over all bits of the input but instead only allows limited information gain about its intermediate states, e.g., the branch in line 10 reveals whether the least significant bit of the intermediate state is 1. Previous work [8] has shown that with a set of manually-crafted non-trivial rules, page-granular side-channel leakage is sufficient to recover one input by assuming knowledge of



(a) Concurrently active states during Athena's path exploration. For readability reasons, only 1000 exploration steps are displayed.



(b) States pruned by Athena due to control- or data-flow mismatches.

Figure 8: Path exploration results of OpenSSL's BEEA implementation: (a) active states, (b) pruned states.

the other input, which is typically publicly known [8, 22, 39]. This case study shows that our approach can automatically infer the rules required to recover the secret input from the BEEA.

Results. Our experiment uses a randomly chosen secret together with the (fixed) public RSA exponent as input, in line with previous work [8, 22, 39]. Athena autonomously recovers the secret input within hours to days, depending on the concrete secret. To evaluate whether the combination of data- and control-flow constraints benefits our approach, we test the experiment with just data- and just control-flow constraints. In both cases, Athena cannot recover the secret input, thus confirming that the combination of both types of constraints is beneficial depending on the target. Figure 8 displays detailed statistics about Athena's path exploration process on the BEEA implementation. Figure 8a shows the number of concurrently active states in the SEE, oscillating between 1 and 10. We observe that the constraints added by the leakage allow the SEE to continuously reduce the number of active states whenever mismatches to the side-channel leakage are detected. Figure 8b shows the overall impact of this, resulting in the pruning of multiple thousand steps during the exploration. Furthermore, Figure 9 highlights how the amount of data-flow constraints increases over time, resulting in multiple thousand constraints added during exploration. As neither the data- nor the control-flow access patterns of the BEEA implementation (cf. Section B) directly depend on its inputs, we conclude that Athena can recover secrets from complex mixed data- and control-flow constraints.

5.5 Attacking RC4 KSA

In this case study, we attack an implementation of the RC4 key-scheduling algorithm (KSA). To the best of our knowl-

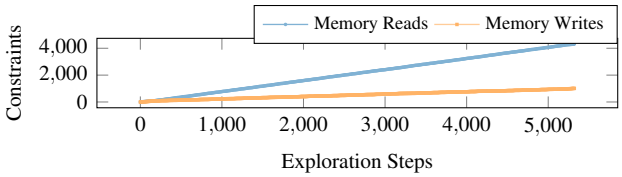


Figure 9: Data-flow constraints added during the path exploration of OpenSSL's BEEA implementation. Constraints come from memory reads and memory writes.

edge, we are the first to show an entire key recovery on this using side-channel leakage. Athena can automatically recover the private key from leakage caused by lookup-table accesses.

Overview. In a typical RC4 KSA implementation, as illustrated in Section C, the internal state is represented as an array of 256 entries. The KSA generates a random key-dependent permutation by iterating over the internal state and swapping the current element with another element based on the key. These accesses lead to leakage exploitable by Athena.

While RC4 is no longer considered secure due to the underlying cryptographic structures [110], it still acts as a good example of secret-dependent data-flow leakage that a side-channel attacker can exploit. While a typical side-channel attacker has to model the everchanging content of `S` to recover the key, Athena can automatically recover the key without an attacker having to investigate the function `KSA`.

As this attack requires a fine-grained leakage, we manually extract a byte-granular data-flow trace that an attacker can create using a fine-grained side-channel attack, e.g., similar to Gyselinck et al. [95] or Hetterich et al. [109]. Still, this demonstrates the applicability of SCASE to previously unexploited targets requiring a fine-grained side-channel attack.

Results. We try to recover 10 randomly generated 256 bit keys from the implementation displayed in Section C. We execute the recovery process on a commodity laptop equipped with an Intel Core i7-1165G7 CPU and 16 GB RAM. For each run, Athena successfully recovers 100 % of the key. The recovery time ranges from 4 min and 29 s to 4 min and 54 s with an average of 4 min and 35 s. For all runs, the time spent in the solving phase was below 10 ms. We conclude that given a fine-grained memory trace, our approach can automatically recover cryptographic keys from applications that leak key bits via their data-flow activity. Further, we show that Athena recovers the key from a side-channel attack that has not been previously exploited without requiring access to expensive computation resources.

5.6 Recovering Poker Cards

To demonstrate SCASE on non-cryptographic targets, we recover cards processed by a poker hand evaluator. We attack the implementation of the TwoPlusTwoHandEvaluator[1], which is based on a poker-hand evaluation algorithm by Paul D. Senzee and Cactus Kev [111, 112].

Overview. While the space of possible poker hands comprises millions of states, all hands can be ranked based on their strength compared to other potential hands [111]. Poker hand evaluators calculate the strength of one or multiple hands, e.g., to determine the winner of a poker game. To speed up the evaluation, Paul D. Senzee and Cactus Kev developed an evaluation strategy that acts as a precomputed hash function [111, 112]. The TwoPlusTwoHandEvaluator is an implementation of this evaluation strategy. In the implementation, the value of a hand is determined by following a pointer chain through a precomputed hash table. Each of the 7 accesses, besides the first one, depends on the result of the previous accesses, and the hash table consists of 32.49 million entries. Hence, manually recovering the complete hand is challenging, as it requires an in-depth understanding of the algorithm.

Results. Our evaluation recovers 10 randomly generated poker hands. We execute the recovery with a commodity laptop equipped with an Intel Core i7-1165G7 CPU and 16 GB RAM and conduct the experiment on byte-granular traces. For each run, we successfully recover 100 % of the cards. Athena spends on average 255 ms, ranging from 250 ms to 280 ms in the path exploration and below 10 ms in the solving phase.

6 Discussion

In the following, we discuss related work and how SCASE and our Athena framework compare to it (Section 6.1). Furthermore, we discuss the limitations of SCASE (Section 6.2).

6.1 Related Work

The following paragraphs discuss previous approaches that aimed to automate secret recovery from side-channel leakage.

Machine Learning for Side-Channel Attack Automation. Prior work has investigated the potential of machine learning for secret recovery from side-channel attacks [33, 34]. Yuan et al. [33] focused on recovering images and text from traces generated using the Intel PIN utility or Prime+Probe [64] via manifold learning. For image recovery, they achieved an accuracy of up to 45.4 %, where a face recognition API determined accuracy. For text recovery, they achieved an accuracy of up to 43.4 % for a next-word predictor. Note that the recovered secrets were only *similar* to the originals and not identical. A later reproduction study by Zhang et al. [34] indicates that Yuan et al. tend to only generate images instead of reconstructing them from actual leakage. They further analyzed the effectiveness of auto encoders in recovering images. While promising for artificial memory traces, their approach was unsuccessful in recovering images from leakage by actual side-channel attacks. GPAM [113] is a deep learning framework targeting physical side-channel

[1] https://github.com/tangentforks/TwoPlusTwoHandEvaluator

attacks on cryptographic algorithms to produce probabilistic results. In contrast, our approach recovers secrets with *certainty* from *generic software-based* side-channel attacks.

Generally, SCASE improves upon machine-learning-based approaches by enabling accurate secret recovery, i.e., our approach can infer complete secrets instead of recreating sufficiently similar secrets, or probabilistically recovering secrets. In contrast to Zhang et al. [34], our approach applies to non-artificial memory traces generated by real-world side-channel attacks. Moreover, SCASE does not require an initial training phase, further reducing the effort for generating attacks.

Symbolic Execution for Side-Channel Attack Automation. Phan et al. [114] automate input generation to maximize an application's side-channel leakage. The approach is tailored towards side-channel attacks requiring multiple runs to leak a secret fully. Their framework can synthesize the next input an attacker needs to query to minimize the remaining entropy for the side-channel attacker. In contrast, our approach targets the orthogonal problem of reconstructing the secret input from the observed leakage. Dubrova [31] demonstrated an attack on AES-128 by combining deep-learning-based power analysis with SAT solving. While their work focuses on power side channels, uses deep learning, and specifically targets AES, our approach targets memory access traces from generic binaries and automatically generates logic constraints.

Differential Fault Analysis. Hu et al. [32] developed an approach to automatically extracting secrets by analyzing the longest recurring sequence of memory accesses. By analyzing the number of memory accesses between the two longest recurring sequences, they can determine whether the secret contains a zero or a one bit. While their approach works on vulnerable programs where leakage stems from branch conditions of loops, our technique makes fewer assumptions about the program's control flow. Hence, our approach applies to a wider range of target applications and side-channel attacks.

6.2 Limitations

The major limitations of SCASE stem from the limitations of symbolic execution. As the SEE needs to emulate the victim application, we require access to the code or the binary of the victim application. Access to code of the victim application is, however, not only in line with with Kerckhoff's principle but also a typical scenario for realistic attack vectors [89]. SCASE computes the relationship between the side-channel leakage and the corresponding secret by solving the constraints created during the exploration of the binary with a SAT solver. Despite being powerful, SAT solvers can not solve arbitrary equations in a reasonable time. While for the applications attacked in this paper, the SAT solving time was negligible (cf. Section 5), it could become infeasible to solve for secrets that have a very complex correlation with the side-channel leakage. Our approach empowers an SEE using the meta information leaked by a side-channel attack. Hence, the amount of information leakage directly correlates with the capability to recover the target application's secret. Thus, SCASE will likely be unsuccessful on a program that does not show any or only minor side-channel leakage. Furthermore, while SCASE significantly reduce the human effort required to recover secrets, human experts are still beneficial for certain targets as they can leverage domain knowledge to come up with more efficient ways to recover a secret.

Athena is focusing on non-interactive single-trace attacks. While Athena also has support for storing and restoring constraints to enable multi-trace attacks, these types of attacks often require specifically crafted inputs, e.g., specific plaintexts, which require manual engineering. Thus, our evaluation consists of victims that can be exploited via a single execution. This excludes interactive attacks, e.g., padding oracles or early-abort string comparisons, where the attacker chooses input values interactively. This limitation can be overcome by adapting the framework to remember constraints for multiple executions and traces of the side-channel attack.

Parts of our evaluation were conducted on byte-granular traces. While side-channel attacks with fine-grained leakage exist [95, 109], this work does not aim to discover novel side-channel attacks nor evaluate the feasibility of these attacks on specific targets. Instead, our approach is based on the leakage stemming from such an attack and demonstrates the potential impact of its leakage. We stress that developers should not rely on the absence of a concrete attack and instead consider every leakage as a potential attack vector, which is in line with the principles of constant-time programming.

7 Conclusion

We proposed SCASE, an approach to automatically infer secrets from side-channel attacks, thus automating the creation of PoC exploits. The key idea behind SCASE is to guide a SEE with side-channel traces. To showcase SCASE, we developed a PoC implementation, Athena, targeting Intel SGX enclaves. Athena recovered a 2048-bit RSA key and a 256-bit secret key of an RC4 KSA. We demonstrated key recovery from OpenSSL's AES S-Box implementation and its binary extended Euclidean algorithm. Furthermore, we recovered the values processed by a poker hand evaluator. We conclude that the technique is versatile and practical in recovering secrets from complex implementations without requiring an in-depth understanding of the victim application.

8 Ethics Considerations

The tool proposed in this work does not discover new vulnerabilities but rather automates the exploitation of known vulnerabilities. Hence, we do not see a direct ethical concern. While it could be used to assist in implementing attack applications, it is intended to be used for defensive purposes

and lowers the barrier for security researchers to evaluate the security of an application. Furthermore, our experiments have been conducted on machines belonging to the authors themselves or their affiliation and do not involve any personal data. Nevertheless, to ensure that Intel is aware of our PoC implementation, we disclosed our work to them.

9 Open Science

To enable the reproducibility of our results and to encourage future work, we released the entire source code of our tool as open-source software, including the memory traces and code samples used in our evaluation to allow for reproducibility of our results.[2]

Acknowledgments

We thank our shepherd and the anonymous reviewers for their feedback and suggestions. We thank Daniel Gruss for his valuable input about this work, especially in the early stages. This work was partly supported by the Semiconductor Research Corporation (SRC) Hardware Security Program (HWS) and by the by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 491039149. Further funding was provided by the Research Fund KU Leuven and by the Cybersecurity Research Program Flanders.

References

[1] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.

[2] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "Smash: Synchronized many-sided rowhammer attacks from javascript." in *USENIX*, 2021.

[3] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[4] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *CCS*, 2015.

[5] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses," in *USENIX Security Symposium*, 2021.

[2]Available at https://github.com/cispa/scase and https://doi.org/10.5281/zenodo.15609410.

[6] D. Gruss, D. Bidner, and S. Mangard, "Practical Memory Deduplication Attacks in Sandboxed JavaScript," in *ESORICS*, 2015.

[7] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C.-m.-t.-n. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *ESORICS*, 2017.

[8] S. Weiser, R. Spreitzer, and L. Bodner, "Single Trace Attack Against RSA Key Generation in Intel SGX SSL," in *AsiaCCS*, 2018.

[9] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX," in *CHES*, 2020.

[10] F. Dall, G. De Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks," in *CHES*, 2018.

[11] P. Qiu, Y. Lyu, H. Wang, D. Wang, C. Liu, Q. Gao, C. Wang, R. Sun, and G. Qu, "Pmuspill: The counters in performance monitor unit that leak sgx-protected secrets," *arXiv:2207.11689*, 2022.

[12] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend," in *USENIX Security Symposium*, 2021.

[13] A. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A False Dependency Attack against Constant-Time Crypto Implementations in SGX," in *CT-RSA*, 2018.

[14] D. Kim, D. Jang, M. Park, Y. Jeong, J. Kim, S. Choi, and B. B. Kang, "SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure," *Computers & Security*, vol. 82, 2019.

[15] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *USENIX Security Symposium*, 2017.

[16] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.

[17] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017.

[18] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *CCS*, 2017.

[19] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache Attacks on Intel SGX," in *EuroSec*, 2017.

[20] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *CCS*, 2018.

[21] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution," in *USENIX Security Symposium*, 2017.

[22] D. Moghimi, J. V. Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled Instruction-Level Attacks on Enclaves for Maximal Key Extraction," in *USENIX Security Symposium*, 2020.

[23] K. Ryan, "Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone," in *CCS*, 2019.

[24] M. Schwarz and D. Gruss, "How Trusted Execution Environments Fuel Research on Microarchitectural Attacks," *IEEE Security & Privacy*, 2020.

[25] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated Discovery of Microarchitectural Side Channels," in *USENIX Security*, 2021.

[26] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.

[27] A. Geimer, M. Vergnolle, F. Recoules, L.-A. Daniel, S. Bardin, and C. Maurice, "A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries," in *SIGSAC*, 2023.

[28] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, ""they're not that hard to mitigate": What cryptographic library developers think about timing attacks," in *SP*, 2022.

[29] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries," in *USENIX Security*, 2018.

[30] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid Prototyping for Microarchitectural Attacks," in *USENIX Security*, 2022.

[31] E. Dubrova, "Solving aes-sat using side-channel hints: A practical assessment," *Cryptology ePrint Archive*, 2024.

[32] L. Hu, F. Zhang, Z. Liang, R. Ding, X. Cai, Z. Wang, and W. Jin, "Faultmorse: An automated controlled-channel attack via longest recurring sequence," *Computers & Security*, 2023.

[33] Y. yuan, Q. Pang, and S. Wang, "Automated Side Channel Analysis of Media Software with Manifold Learning," in *USENIX Security*, 2022.

[34] Z. Zhang, Z. Lai, and U. Parampalli, "R+R: Demystifying ML-Assisted Side-Channel Analysis Framework: A Case of Image Reconstruction," in *ACSAC*, 2024.

[35] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *S&P*, 2015.

[36] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *Workshop on System Software for Trusted Execution*, 2017.

[37] The angr Project contributors, "angr," 2024. [Online]. Available: https://github.com/angr/angr

[38] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *S&P*, 2016.

[39] A. C. Aldaya and B. B. Brumley, "When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA," *IACR Cryptology ePrint Archive*, 2020. [Online]. Available: https://eprint.iacr.org/2020/055

[40] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *arXiv preprint*, 2011.

[41] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *CRYPTO*, 2010.

[42] E. Stefanov, M. v. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," *JACM*, 2018.

[43] J. Wichelmann, A. Rabich, A. Pätschke, and T. Eisenbarth, "Obelix: Mitigating side-channels through dynamic obfuscation," in *S&P*, 2024.

[44] S. Aga and S. Narayanasamy, "Invisipage: oblivious demand paging for secure enclaves," in *International Symposium on Computer Architecture*, 2019.

[45] P. Zhang, C. Song, H. Yin, D. Zou, E. Shi, and H. Jin, "Klotski: Efficient obfuscated execution against controlled-channel attacks," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[46] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "Dr. SGX: automated and adjustable side-channel protection for SGX using data location randomization," in *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[47] W. Kosasih, Y. Feng, C. Chuengsatiansup, Y. Yarom, and Z. Zhu, "SoK: Can We Really Detect Cache Side-Channel Attacks by Monitoring Performance Counters?" in *AsiaCCS*, 2024.

[48] D. Weber, L. Niemann, L. Gerlach, J. Reineke, and M. Schwarz, "No Leakage Without State Change: Repurposing Configurable CPU Exceptions to Prevent Microarchitectural Attacks," in *ACSAC*, 2024.

[49] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016.

[50] M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *ESSoS*, 2016.

[51] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," in *NDSS*, 2017.

[52] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu," in *AsiaCCS*, 2017.

[53] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *USENIX Annual Technical Conference (ATC)*, 2018.

[54] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic side-channel resistance using efficient control and data flow linearization," in *SIGSAC*, 2021.

[55] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[56] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A flexible, constant-time programming language," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 69–76.

[57] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-assurance and high-speed cryptography," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[58] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *CCS*, 2017.

[59] S. Constable, J. Van Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein, "Aex-notify: Thwarting precise single-stepping attacks through interrupt awareness for intel sgx enclaves," in *USENIX*, 2023.

[60] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: Closing controlled channels with self-paging enclaves," in *European Conference on Computer Systems (EuroSys)*, 2020.

[61] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[62] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[63] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *CCS*, 2021.

[64] C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.

[65] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium*, 2018.

[66] S. Bhattacharya and D. Mukhopadhyay, "Who watches the watchmen?: Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms," *Cryptology ePrint Archive, Report 2015/621*, 2015.

[67] O. Acıiçmez, J.-P. Seifert, and c. K. Koç, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007.

[68] T. Schlüter, A. Choudhari, L. Hetterich, L. Trampert, H. Nemati, A. Ibrahim, M. Schwarz, C. Rossow, and N. O. Tippenhauer, "FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers," in *CCS*, 2023.

[69] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *NDSS*, 2020.

[70] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud," in *USENIX Security Symposium*, 2012.

[71] T. Rokicki, C. Maurice, and M. Schwarz, "CPU Port Contention Without SMT," in *ESORICS*, 2022.

[72] L. Yan, Y. Guo, X. Chen, and H. Mei, "A Study on Power Side Channels on Mobile Devices," in *Symposium on Internetware*, 2015.

[73] A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard, "Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels," in *USENIX Security*, 2023.

[74] F. Rauscher, C. Fiedler, A. Kogler, and D. Gruss, "A systematic evaluation of novel and existing cache side channels," in *NDSS*, 2025.

[75] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *ESORICS*, 2019.

[76] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical cache attacks from the network," in *S&P*, 2020.

[77] M. Schwarzl, P. Borrello, G. Saileshwar, H. Müller, M. Schwarz, and D. Gruss, "Practical Timing Side-Channel Attacks on Memory Compression," in *S&P*, 2023.

[78] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, 1976.

[79] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008.

[80] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *S&P*, 2020.

[81] F. Alder, L.-A. Daniel, D. Oswald, F. Piessens, and J. Van Bulck, "Pandora: Principled symbolic validation of intel sgx enclave runtimes," in *S&P*, 2024.

[82] T. Cloosters, M. Rodler, and L. Davi, "{TeeRex}: Discovery and exploitation of memory corruption vulnerabilities in {SGX} enclaves," in *USENIX Security Symposium*, 2020.

[83] P. Antonino, W. A. Woloszyn, and A. Roscoe, "Guardian: Symbolic validation of orderliness in sgx enclaves," in *CCSW*, 2021.

[84] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." in *NDSS*, 2016.

[85] Y. Wang, S. Sheng, and Y. Wang, "A systematic literature review on smart contract vulnerability detection by symbolic execution," in *International Conference on Blockchain and Trustworthy Systems*, 2024.

[86] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "Specusym: Speculative symbolic execution for cache timing leak detection," in *ACM/IEEE International Conference on Software Engineering*, 2020.

[87] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle, and V. Zimmer, "Symbolic execution for {BIOS} security," in *WOOT*, 2015.

[88] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "{FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *USENIX*, 2013.

[89] V. Costan and S. Devadas, "Intel SGX Explained," *Cryptology ePrint Archive, Report 2016/086*, 2016.

[90] Intel, "Get Started with the SDK," 2019. [Online]. Available: https://software.intel.com/en-us/sgx/sdk

[91] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of sgx and countermeasures: A survey," *CSUR*, 2021.

[92] A. Nilsson, P. Nikbakht Bideh, and J. Brorsson, "A survey of published attacks on intel sgx," *arXiv preprint*, 2020.

[93] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security*, 2018.

[94] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[95] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx, "Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution," in *ESSoS*, 2018.

[96] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *S&P*, 2020.

[97] Intel, "Intel Xeon scalable platform built for most sensitive workloads," October 2020. [Online]. Available: https://www.intel.com/content/www/us/en/newsroom/news/xeon-scalable-platform-built-sensitive-workloads.html

[98] ——, "Intel Trust Domain Extensions," 2023. [Online]. Available: https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf

[99] L. Wilke, F. Sieck, and T. Eisenbarth, "TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX," in *CCS*, 2024.

[100] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "Sev-step: A single-stepping framework for amd-sev," 2023.

[101] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based Fault Injection using Selective State Reset," in *USENIX Security*, 2024.

[102] R. Spreitzer and T. Plos, "Cache-Access Pattern Attack on Disaligned AES T-Tables," in *COSADE*, 2013.

[103] N. Lawson, "Side channel attacks on cryptographic software," *IEEE Security & Privacy*, vol. 7, no. 6, pp. 65–68, 2009.

[104] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels," in *USENIX Security*, 2023.

[105] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *CSUR*, 2018.

[106] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *CCS*, 2016.

[107] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think," in *USENIX Security Symposium*, 2018.

[108] E. Oswald, S. Mangard, N. Pramstaller, and V. Rijmen, "A side-channel analysis resistant description of the aes s-box," in *Fast Software Encryption*, 2005.

[109] L. Hetterich, F. Thomas, L. Gerlach, R. Zhang, N. Bernsdorf, E. Ebert, and M. Schwarz, "ShadowLoad: Injecting State into Hardware Prefetchers," in *ASPLOS*, 2025.

[110] P. Jindal and B. Singh, "RC4 Encryption - A Literature Survey," in *ICICT*, 2014.

[111] S.-g. Kim and Y.-G. Kim, "A learning ai algorithm for poker with embedded opponent modeling," *International Journal of Fuzzy Logic and Intelligent Systems*, 2010.

[112] L. F. Teófilo, L. P. Reis, and H. L. Cardoso, "Computing card probabilities in texas hold'em," in *CISTI*, 2013.

[113] E. Bursztein, L. Invernizzi, K. Král, D. Moghimi, J.-M. Picod, and M. Zhang, "Generalized power attacks against crypto hardware using long-range deep learning," *CHES*, 2024.

[114] Q.-S. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, "Synthesis of adaptive side-channel attacks," in *CSF*, 2017.

A Athena Usage Example

Listing 2 shows an example of how to use Athena to recover a secret key from a victim function. Only the code parts highlighted in orange and written in cursive require changes when adapting the code to a different target.

```
athena_framework = athena.AthenaFramework(
  ENCLAVE_PATH, TARGET_ECALL, TARGET_FUNC,
  enable_control_flow_tracing=True,
  control_flow_tracefile=CFTRACE_FILE,
  enable_data_flow_tracing=True,
  data_flow_tracefile=DFTRACE_FILE,
  base_addr=ENCLAVE_BASE_ADDR)

s = athena_framework.get_initial_state()

# Victim function call layout:
# RDI: pointer to secret key
# RSI: len(secret key)

l = s.regs.rsi.concrete_value
scrt = s.solver.BVS("secret", l * 8)
s.memory.store(s.regs.rdi.concrete_value, scrt)

athena_framework.set_initial_state(s)

athena_framework.run()
solution = athena_framework.solve(scrt)
```

Listing 2: Example usage of Athena. Only the *highlighted* are target-specific, as they annotate the value to be recovered. In this case, the `RDI` register points to the secret.

B OpenSSL's Binary Extended Euclidean Algorithm

Listing 3 shows the implementation of OpenSSL's binary extended Euclidean algorithm (BEEA).

C RC4 Key Scheduling Algorithm

Listing 4 shows the implementation of the RC4 key scheduling algorithm (KSA) used in our evaluation.

```
static BIGNUM *euclid(BIGNUM *a, BIGNUM *b) {
  BIGNUM *t;
  int shifts = 0;
  bn_check_top(a);
  bn_check_top(b);

  while (!BN_is_zero(b)) {
    if (BN_is_odd(a)) {
      if (BN_is_odd(b)) {
        if (!BN_sub(a, a, b))
          goto err;
        if (!BN_rshift1(a, a))
          goto err;
        if (BN_cmp(a, b) < 0) {
          t = a; a = b; b = t;
        }
      } else {
        if (!BN_rshift1(b, b))
          goto err;
        if (BN_cmp(a, b) < 0) {
          t = a; a = b; b = t;
        }
      }
    } else {
      if (BN_is_odd(b)) {
        if (!BN_rshift1(a, a))
          goto err;
        if (BN_cmp(a, b) < 0) {
          t = a; a = b; b = t;
        }
      } else {
        if (!BN_rshift1(a, a))
          goto err;
        if (!BN_rshift1(b, b))
          goto err;
        shifts++;
      }
    }
  }
  if (shifts) {
    if (!BN_lshift(a, a, shifts))
      goto err;
  }
  bn_check_top(a);
  return (a);
err:
  return (NULL);
}
```

Listing 3: OpenSSL's implementation of the binary extended euclidian algorithm.

```
for (int i = 0; i < 256; i++)
  S[i] = i;
int j = 0;
for (int i = 0; i < 256; i++) {
  j = (j + S[i] + key[i % keylen]) % 256;
  swap(S[i], S[j]);
}
```

Listing 4: RC4 Key Scheduling Pseudocode.

D Jump-Table Example Implementation

Listing 5 shows the toy victim that we attack in Section 5.1.1.

```
unsigned char dec_to_hex[] = "0123456789abcdef";
unsigned char hex_to_dec_lut[256];

unsigned char hex_to_dec(unsigned char c) {
    return hex_to_dec_lut[c];
}

unsigned char buffer[KEYSIZE] = {0};
unsigned int buffer_pos = 0;

void func_0() { buffer[buffer_pos++] = '0'; }
void func_1() { buffer[buffer_pos++] = '1'; }
[...]
void func_f() { buffer[buffer_pos++] = 'f'; }

struct lut_entry {
    void* func;
    char padding[PADDING];
};

struct lut_entry __attribute__((aligned(4096)))
lut[16] = {
    {&func_0, {0}}, {&func_1, {0}},
    [...], {&func_f, {0}},
};

void __attribute__((aligned(4096))) victim(
  unsigned char* key, int len,
  struct lut_entry* lut) {
    for (int i = 0; i < len; i++) {
        int idx = hex_to_dec(key[i]);
        ((void(*)())lut[idx].func)();
    }
}
```

Listing 5: Toy victim program exposing control- and data-flow side-channel leakage.

USENIX Security '25 Artifact Appendix: SCASE: Automated Secret Recovery via Side-Channel-Assisted Symbolic Execution

Daniel Weber
CISPA Helmholtz Center
for Information Security

Lukas Gerlach
CISPA Helmholtz Center
for Information Security

Leon Trampert
CISPA Helmholtz Center
for Information Security

Youheng Lü
SCHUTZWERK GmbH

Jo Van Bulck
DistriNet, KU Leuven

Michael Schwarz
CISPA Helmholtz Center
for Information Security

A Artifact Appendix

A.1 Abstract

This paper proposes SCASE, a novel methodology for inferring secrets from an opaque victim binary using symbolic execution guided by a concrete side-channel trace. Our key innovation is utilizing the memory accesses observed in the side-channel trace to prune the symbolic-execution space, thus avoiding state explosion. To demonstrate the effectiveness of our approach, we introduce Athena, a proof-of-concept framework to recover secrets from Intel SGX enclaves via controlled channels automatically. We show that Athena can automatically recover the 2048-bit secret key of an enclave running RSA and the 256-bit key from an RC4 KSA implementation. Furthermore, we demonstrate key recovery of OpenSSL's 256-bit AES S-Box implementation and recover the inputs to OpenSSL's binary extended Euclidean algorithm. To demonstrate the versatility of our approach beyond cryptographic applications, we further recover the input to a poker-hand evaluator. Our findings indicate that constraining symbolic execution via side-channel traces is an effective way to automate software-based side-channel attacks without requiring an in-depth understanding of the victim application.

A.2 Description & Requirements

The majority of our artifact only requires a Linux installation supporting Python 3 and a recent *angr*[1] installation. For recovering the 2048-bit RSA key, a machine with at least 64 GB of RAM is required. The only (optional) hardware requirement is required to regenerate the memory traces for the 2048-bit RSA key. For regenerating the memory traces for the 2048-bit RSA key, which we consider an optional experiment as we provide the traces alongside our artifact, an Intel SGX machine is required.

[1] https://github.com/angr/angr

A.2.1 Security, privacy, and ethical concerns

Our framework does not pose any security, privacy, or ethical concerns, as it only executes Python code. Nevertheless, the (not required) execution of our *ptrace tracer* requires the user to disable ASLR. For this, scripts exist to dis- and enable ASLR in the corresponding folder. Furthermore, our artifact contains real-world code used as meaningful victim applications. While the source code of these applications is publicly available, we cannot guarantee that the code is free of security, privacy, or ethical concerns. To the best of our knowledge, the code does not contain any such concerns.

A.2.2 How to access

The artifact is publically available on GitHub `https://github.com/cispa/scase`.
Furthermore, the artifact is archived on Zenodo with DOI `https://doi.org/10.5281/zenodo.15609410`. Note that, when fetched from Zenodo, the file `scase-traces-*.zip` needs to be extraced to `./traces`.

A.2.3 Hardware dependencies

Our artifact requires a Linux machine with at least 62 GB of allocatable RAM. The optional regeneration of the memory traces for the RSA key requires an Intel SGX machine.

A.2.4 Software dependencies

Our installation instructions are written for Ubuntu 22.04, but the artifact should work on most recent Linux distributions. We require Python 3.10 or higher, an *angr* installation, and at least 62 GB of allocatable RAM.

A.2.5 Benchmarks

Our artifact contains the memory traces used during the evaluation of our framework. These memory traces are CSV files

containing the memory accesses of the victim application. While we provide instructions on regenerating the memory traces, users are free to use the provided memory traces.

A.3 Set-up

A.3.1 Installation

After cloning the repository, create a virtual environment and install the required dependencies described under "Dependencies" in the `athena/README.md` file.

A.3.2 Basic Test

To run a basic test, execute the following command (after loading the virtual environment and from within the folder `./athena`): `python3 ./hex_elf.py`. Upon successful execution, the output should contain `Encoded solution: 8`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Athena can automatically recover secrets for varying granularities of memory traces, which are not recovered without memory trace guidance. This is shown by Experiment E1 (Section 5.1.1) and Experiment E6 (Sections 5.1.2 and 5.2).

(C2): Athena can recover an AES key from OpenSSL's S-Box implementation and the input of OpenSSL's binary extended Euclidean algorithm implementation. This is shown by Experiment E2 (Section 5.3) and Experiment E3 (Section 5.4).

(C3): Athena can recover the input to an RC4 KSA implementation. This is shown by Experiment E4 (Section 5.5).

(C4): Athena can recover the input to non-cryptographic applications, as demonstrated by its applicability to the TPT hand evaluator. This is shown by Experiment E5 (Section 5.6).

A.4.2 Experiments

To ease the reproduction of our results, we provide precompiled victim applications and memory traces in the `traces` folder. The Athena framework uses angr to perform symbolic execution on these binaries. Thus, the scripts, also referred to as *Athena wrappers* in the latter, never actually execute these binaries but rather emulate their execution. Experiment E7 (optional) describes how to recompile the victim applications and regenerate the memory traces. Note that the regeneration of the traces requires an Intel SGX machine with SGX-Step[2] installed, which is not required for the other experiments.

[2] https://github.com/jovanbulck/sgx-step

(E1): [Jump-Table Example] *[30 human-minutes + 2 compute-hours]:* This experiment shows Athena recovering the key from our Jump-table toy victim.
How to: The Athena wrapper for this can be found in `./athena/hex_elf_eval.py`. Our results, which are used in Figures 5 and 6 of the paper, can be found in `traces/jump-table/`. The victim program's code can be found in `./victim-programs/jump-table/`
Preparation: Traverse to the folder `./athena` and load the installation's venv.
Execution: Execute `python3 ./hex_elf_eval.py` and `python3 ./hex_elf_eval_noprune.py`. This generates subfolders with the following structure: `eval_data_g_<bytelength>_<option>`. When `<option>` equals `noprune`, the symbolic execution is not pruned by the side-channel trace.
Results: The resulting statistics are contained in the file `statistics.csv` for each subfolder. These (accumulated) numbers correspond to the paper's Figures 5 and 6. The most noteworthy result is that the `<...>_noprune` subfolders should indicate that the value could not be recovered. This can be confirmed by checking the column `incorrect_bytes` in the corresponding `statistics.csv` file.

(E2): [AES S-Box] *[30 human-minutes + 30 compute-minutes]:* This experiment shows Athena recovering the symmetric key from OpenSSL's AES S-Box implementation.
How to: The Athena wrapper for this can be found in `./athena/aes_openssl.py`. The victim program's code can be found in `./victim-programs/openssl-aes-sbox/`.
Preparation: Traverse to the folder `./athena` and load the installation's venv.
Execution: Execute `python3 ./aes_openssl.py`. This recovers the 256-bit AES key and prints whether the key is correct. The key can be verified by checking the file `./victim-programs/openssl-aes-sbox/victim.c`. The command `python3 ./aes_openssl_eval.py` can be used to execute the experiment for different memory trace granularities.
Results: The execution of `aes_openssl.py` should print that the key was successfully recovered. The execution of `aes_openssl_eval.py` creates two folders `eval_data_g01` and `eval_data_g01_dfonly`, where the latter restricts the guidance to the data-flow trace. Both folders contain a file `statistics.csv`, which contains the detailed statistics (similar to E1). The number of incorrect bytes for e should align with the paper's Table 1.

(E3): [BEEA] *[30 human-minutes + 1 compute-hour]:* This experiment shows Athena recovering the key from OpenSSL's binary extended Euclidean algorithm

(BEEA) implementation.

How to: The Athena wrapper for this can be found in `./athena/beea_openssl.py`. The victim program's code can be found in `./victim-programs/openssl-beea/`.

Preparation: Traverse to the folder `./athena` and load the installation's venv.

Execution: The experiment can be executed by running `python3 ./beea_openssl.py`. This recovers the second argument to `BN_gcd`. Note that this takes around 10-14 hours to finish.

Results: The execution of `aes_openssl.py` should print the key contained in the variable `p_min_one` from `./victim-programs/openssl-beea/main.c`.

(E4): [RC4-KSA] *[30 human-minutes + 5 compute-minutes]:* This experiment shows Athena can recover the RC4 key from a key scheduling algorithm (KSA) implementation.

How to: The Athena wrapper for this can be found in `./athena/rc4_elf.py`. The victim program's code can be found in `./victim-programs/rc4-ksa/`.

Preparation: Traverse to the folder `./athena` and load the installation's venv.

Execution: Execute `python3 ./rc4_elf.py`. This recovers the key from `victim-programs/rc4-ksa/main.c`.

Results: The execution of `rc4_elf.py` should print the key recovered from the binary. This key should match with the content of `./victim-programs/rc4-ksa/key.hex`.

(E5): [TPT Hand Evaluator] *[30 human-minutes + 2 compute-minutes]:* This experiment shows Athena can recover secrets from non-cryptographic applications, in this case, the input to a poker-hand evaluator.

How to: The Athena wrapper for this can be found in `./athena/poker_elf.py`. The victim program's code can be found in `./victim-programs/tpt-hand-evaluator/`.

Preparation: Traverse to the folder `./athena` and load the installation's venv. Additionally, unpack the binaries contained in the tar balls in `./traces/tpt-hand-evaluator/`. For this, run `find . -name '*.tar.gz' -exec tar -xzvf {} \;` from within the folder.

Execution: The experiment can be executed by running `python3 ./poker_elf.py`. This recovers the card array from `victim-programs/tpt-hand-evaluator/test.c`.

Results: The execution of `poker_elf.py` should print the array contained in the variable `cards` in `./victim-programs/tpt-hand-evaluator/test.c`. As we executed this program multiple times with varying inputs, the folder `./traces/tpt-hand-evaluator/` does not only contain the binaries and memory traces but also the expected outcomes for each input. The combinations of input and expected output are enumerated. For example, the files `cftrace1.csv`, `dftrace1.csv`, `victim1`, and `solution1.txt` belong together.

(E6): [Intel SGX Square+Multiply Enclave] *[2 human-hours + 10 compute-minutes]:* This experiment recovers a 2048-bit key from an Intel SGX enclave implementation of a toy Square+Multiply algorithm.

How to: The Athena wrapper for this can be found in `./athena/sm_enclave.py`. The victim program's code can be found in `./victim-programs/square-multiply-enclave/`.

Preparation: Traverse to the folder `./athena` and load the virtual environment from the installation.

Execution: The experiment can be executed by running `python3 ./sm_enclave.py`.

Results: The printed key should match the variable `secret` contained in the file `square-multiply-enclave/encl.c`.

(E7 - Optional): [Victim Recompilation and Trace Regeneration] *[4 human-hours + 2 compute-hours]:* This experiment is about recompiling the victim applications and regenerating the memory traces provided in `./traces/`.

How to: For each victim application, the corresponding folder contains a `README.md` file with complication instructions, e.g., the instructions to recompile the Jump-Table example are found in `./victim-programs/jump-table/README.md`.

Preparation: The Jump-Table example does not require any special preparation besides a `gcc` installation. The AES and BEEA examples require the corresponding OpenSSL library to be compiled. The hand evaluator requires a `g++` installation. The Square+Multiply Enclave requires Intel SGX and SGX-Step installed.

Execution: Follow the instructions in `victim-programs/<victim>/README.md` to recompile the victim application and regenerate the memory traces.

Results: The binaries and traces should match the ones provided in `./traces/<victim>`. Note that due to varying environments, it is unlikely that the binaries and resulting traces are identical. One can copy the resulting binaries and memory traces to the corresponding folder in `./traces/` rerun the corresponding experiment.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.