A Deep Dive into Function Inlining and its Security Implications for ML-based Binary Analysis

Omar Abusabha, Jiyong Uhm, Tamer Abuhmed, Hyungjoon Koo*
Sungkyunkwan University, South Korea
404970@g.skku.edu, jiyong423@g.skku.edu, tamer@skku.edu, kevin.koo@skku.edu

***Abstract*—A function inlining optimization is a widely used transformation in modern compilers, which replaces a call site with the callee's body in need. While this transformation improves performance, it significantly impacts static features such as machine instructions and control flow graphs, which are crucial to binary analysis. Yet, despite its broad impact, the security impact of function inlining remains underexplored to date. In this paper, we present the first comprehensive study of function inlining through the lens of machine learning-based binary analysis. To this end, we dissect the inlining decision pipeline within the LLVM's cost model and explore the combinations of the compiler options that aggressively promote the function inlining ratio beyond standard optimization levels, which we term *extreme inlining*. We focus on five ML-assisted binary analysis tasks for security, using 20 unique models to systematically evaluate their robustness under extreme inlining scenarios. Our extensive experiments reveal several significant findings: i) function inlining, though a benign transformation in intent, can (in)directly affect ML model behaviors, being potentially exploited by evading discriminative or generative ML models; ii) ML models relying on static features can be highly sensitive to inlining; iii) subtle compiler settings can be leveraged to deliberately craft evasive binary variants; and iv) inlining ratios vary substantially across applications and build configurations, undermining assumptions of consistency in training and evaluation of ML models.**

I. Introduction

Today, most applications are distributed in the form of executable binaries containing low-level machine instructions and embedded data. Modern compilers produce these binaries via a sophisticated compilation pipeline, which involves a broad spectrum of optimizations. Among them, function inlining is a well-established optimization technique that replaces a function call site with the body of the callee. While this may appear to be a trivial transformation, determining an optimal inlining strategy is challenging – comparable to the NP-complete Knapsack problem [94] in complexity.

When the source code is unavailable, binary reverse engineering (*i.e.,* binary reversing) is commonly employed to comprehend the inner workings of a binary, unveiling hidden or unknown functionality. The binary reversing can be performed either statically [105], [53], [2], [6] (*i.e.,* analysis without executing a binary) or dynamically [31], [90], [108]. Since the disassembled instructions in binaries lack high-level information (*e.g.,* function names, variable names, or types), traditional approaches extract a wide range of features, such as numeric patterns, control and data flows, and instruction sequences. Such features play a crucial role in security tasks, including function boundary detection [11], [84], binary code similarity detection (BCSD) [28], [5], [85], [75], [104], [100], vulnerability detection [38], [96], malware analysis [6], malware family classification [60], [21], and crash root cause analysis [82].

Recent advances integrate machine learning (ML) (*e.g.,* deep learning) techniques to analyze binaries in a high-dimensional feature space. A vast amount of research in the literature have demonstrated the effectiveness of this new direction, guiding a reverse engineer by inferring function names [55], [41], [24], variable names [41], [15] and types [41], [15], and even recovering decompiled code [36], [50], in addition to the tasks listed above.

However, function inlining as a common compiler optimization can *substantially distort static features* for reverse engineering. Inlining merges the callee's instructions into the caller, which can drastically alter machine code and control flow structures. These effects are amplified with nested inlining or when additional optimizations follow. Consequently, static features may no longer be reliable for decision-making in ML models when aggressive inlining is applied (on purpose).

Although the substantial impact of function inlining is well known, an *in-depth study of its security implications on ML-based models* (*e.g.,* to what extent) has remained underexplored in the existing literature. While several works have considered function inlining [43], [44], [14], [28], others have occasionally misrepresented [8], underestimated [53], [56], [64], [32], heuristically addressed [14], [28], [107], [106] it, or paid limited attention [86], [32], [103], [33], [105], [46], [83], [49], [84], [11]. To the best of our knowledge, this is the first study that delves into ① uncovering the security implications of function inlining in the context of modern ML-based models; ② investigating the entire decision-making pipeline of inlining a function, including the internal workings of the compiler's cost model; ③ exploring a broad set of (often overlooked) compiler options that affect function inlining decisions; and ④ identifying a particular combination of options that can increase the inlining ratio (*e.g.,* up to 79.64% in our experiments). Our study is grounded in the LLVM compiler toolchain, which offers well-modularized building blocks. We leverage debugging information and intermediate compiler-producing outputs to construct the ground truth.

* Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2026
23–27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241872
www.ndss-symposium.org

LLVM's inlining process, in essence, follows a structured pipeline: the frontend (*i.e.,* Clang) and the middle end (*i.e.,* Opt) apply a series of analyses and transformation passes on intermediate representations. The CGSCC (Call-Graph Strongly Connected Components) pass manager internally coordinates inlining and other call-graph-based optimizations across strongly connected components (*i.e.,* sets of mutually reachable functions). The function inlining pass, as part of CGSCC, evaluates each function candidate with a cost model to compare a function's inlining cost with a threshold. This pass is applied iteratively following each update to the call graph. In the meantime, a multitude of compiler options (Table II) can affect this decision.

In this work, we examine the potential misuse of function inlining as a means of circumventing ML-based security models. We identify a specific configuration of compiler options that aggressively increases the inlining ratio, a strategy we term *extreme inlining*. Notably, we demonstrate how this otherwise legitimate compiler mechanism can be exploited by adversaries to introduce *deliberate mutations via extreme inlining*, thereby enabling evasion of both discriminative and generative ML-based security models. Our threat model assumes a standard build process repurposed for malicious ends without the compiler's modification or obfuscation. Additionally, we revisit varying inlining scenarios, clarifying subtleties in function inlining practices.

To validate our claims, we conduct extensive experiments by defining nine research questions from two different perspectives. First, we explore how function inlining affects ML-assisted security applications with the following five tasks by training 20 unique models (Section VI-A): ① binary code similarity detection, ② function symbol name prediction, ③ malware detection, ④ malware family classification, and ⑤ vulnerability detection. Second, we evaluate the function inlining optimization itself in terms of its ratio and further impacts with the following four questions (Section VI-B): ⑥ function inlining ratio and optimization levels across different applications, ⑦ the ratio and the (frontend) compiler options, ⑧ the ratio and a combination of (middle end) compiler options toward extreme inlining, and ⑨ the variation of static features in a binary according to (extreme) inlining.

Our key findings indicate that ① function inlining (itself), while intended as a benign optimization, can be exploited by an adversary to evade both discriminative and generative ML models, particularly those that rely on static features and are sensitive to such transformations; ② subtle inlining configurations that affect a decision pipeline can be deliberately manipulated to produce evasive binary variants with ease; and ③ inlining ratios can vary significantly depending on application-specific factors (*e.g.,* programming style) and build settings (*e.g.,* optimization levels, compiler options), offering ample flexibility for generating diverse mutations.

The following summarizes our contributions.

- To the best of our knowledge, this is the first comprehensive study that focuses solely on function inlining. We investigate the complex decision-making process for function inlining within the LLVM compiler toolchain, diving into the inner heuristics, compiler options, and the cost model for inlining.

TABLE I: Special LLVM options to emit intermediate outputs of function inlining.

Option	**Description**
`-Rpass=inline`	Shows inlining information (*e.g.,* success, failure)
`-Rpass-analysis=inline`	Shows inlining analysis (*e.g.,* cost, threshold)
`-Rpass-missed=inline`	Shows missed inlining (*e.g.,* heuristics, high cost)

- We clarify the misleading subtleties in function inlining practices, which have been underestimated or overlooked.
- We present extreme inlining with a different combination of compiler options, and how it can distort static features (*e.g.,* machine code, control flow graphs).
- We conduct an extensive study on security implications by investigating how (extreme) inlining affects ML-assisted binary reversing tasks.

To facilitate future research, we release our datasets, analysis, and models as open source [1].

II. Background

LLVM Architecture and Intermediate Outputs. LLVM (Low-level virtual machine) [87] is a collection of compiler toolchains, which provides modular building blocks for both the frontend and backend. Its core design centers around an intermediate representation (IR), which enables support for various programming languages at the frontend, analysis and transformation of IRs at the middle end, and target-specific code generation at the backend. LLVM provides numerous options for emitting intermediate results during transformations [2]. In this work, we adopt three special options in LLVM (Table I) for gathering diagnostic reports on function inlining. Although the outputs may be imperfect [17], we harness them to reveal how a cost model internally determines whether a function can be inlined through complex computations (*i.e.,* cost, threshold). As a final note, we explore the LLVM source for in-depth analysis of its function inlining optimization.

Symbols and Linkage Types. The compiler maintains several different linkage types that define the visibility of a symbol when emitting a binary, which determine how they can be accessed across different modules. These types are crucial to handling symbol visibility, code optimization, and symbol collision. First, a symbol with an external linkage is supposed to be visible outside of a module for being referenced by other modules, including functions and variables without the `static` keyword or with `extern`. A function is declared with an external linkage by default [91]. Second, a symbol with an internal linkage is only visible within a module, which cannot be referenced from others (but avoid name collisions). It is noted that static functions are considered as unused global symbols (thus safely removed [23]) once they are inlined. Third, a symbol with a private linkage is analogous to internal ones, with an additional constraint that even link time optimization (LTO) cannot make it visible to other translation units. Lastly, a symbol with a weak linkage allows for multiple definitions of the identical symbol across different modules, which the linker chooses from afterward.

[1] https://doi.org/10.5281/zenodo.17759528

[2] GCC offers the `-fdump-ipa-inline` option for determining function inlining in a single-source code [43] alone.

This flexibility renders the multiple symbols in need present for inline functions or template instantiations in C++.

DWARF Information. Debugging With Attributed Record Formats (or DWARF for short) [19] defines a structured data format that contains essential information for debugging such as mappings between source and machine code. A debugging information entry (DIE) enables one to create a low-level representation of a source program, each of which includes an identifying tag and a series of attributes. The tag specifies the class of the entry while the attributes define its specific characteristics. Table VIII in Appendix enumerates the tag and attributes that are associated with the inlining behavior of a function in DWARF. The information can be generated during compilation with the `-g` option. As DWARF provides a unified approach that is orthogonal to the programming language and its underlying structure, it has been widely adopted to obtain the ground truth of an executable binary. Likewise, this work also utilizes DWARF v5 [19] to obtain the ground truth of function inlining (Section VII).

Benefits and Downsides of Function Inlining. A function inlining optimization involves trade-offs. The evident benefit of inlining is to reduce the performance overhead of a function invocation by eliminating additional stack-relevant operations, such as (re)storing a value(s) to a register(s) for passing a parameter(s) and the prologue and epilogue of a function for adjusting stack and base pointers. Next, function inlining may increase instruction cache locality (*e.g.,* an inlined function in a loop). Besides, function inlining provides a chance for further optimization(s). Meanwhile, locating duplicate codes (*i.e.,* instructions) multiple times inevitably grows the size of an executable binary. Another downside would be likely to increase the pressure of allocating more registers due to their longer liveness and more loop invariants. Lastly, a function inlining process increases compilation time.

III. Motivation and Threat Model

Function Inlining in Prior Work. In surveying prior work, we discover that function inlining has often been misrepresented [8], underestimated [53], [56], [64], [32], heuristically addressed [14], [28], [107], [106] it, or paid limited attention [86], [32], [103], [33], [105], [46], [83], [49], [84], [11]. For instance, Alves-Foss *et al.* [8] misrepresent the implications of inlining by stating that "*if a source code function is inlined, it is no longer a function, and an analysis tool should not claim it found that function within the binary*"; however, inlined functions may still appear in the binary. Binkit [53] and discovRE [32] underestimate the pervasiveness of inlining by applying the `-fno-inline` option to prevent inlining. Besides, the `always_inline` directive can override this setting. Meanwhile, Dispatch [56] examines a case where a large trigonometric function (*e.g.,* `tan`) in firmware avoids function inlining to allow for handy vulnerability patching. Many existing works, including those focused on tasks like bug discovery, code search, and code similarity detection [86], [103], [33], [105], [46], [83], [49] do not carefully examine the impact of function inlining, failing to account for its potential influence on code semantics. Similarly, XDA [84] and BYTEWEIGHT [11] rely on function prologues and epilogues, without thoroughly discussing the impact of inlining. Dream(++) [107], [106] addresses inlining heuristically by building signature databases to recognize (known) commonly inlined library functions such as `strcpy`, `strlen` and `strcmp`. On the other hand, CI-Detector [44] and Jia *et al.* [43] underscore handling inlining directives. Collberg *et al.* [18] describe inlining as an effective and practical obfuscation technique because it removes procedural abstraction from the program. Meanwhile, FUNCRE [4] addresses inlined library functions consistently across various optimization levels but encounters challenges when multiple consecutive library functions are inlined together. This work focuses on the unique impact of inlining, such as its ability to alter call graphs, control-flow graphs, and function boundaries.

Threat Model and Assumptions. Our threat model assumes that a benign, standard build process can be repurposed for malicious ends. Under this regime, attackers need not rely on obfuscation to evade detection; they can simply recompile the same code under different optimization strategies or build configurations. We focus on *realistic*, flag-level control of inlining (*e.g.,* optimization levels and inliner parameters) without modifying the compiler's internal logic. Notably, this assumption differs from traditional obfuscation, which involves deliberate, nonstandard transformations that introduce different trade-offs and detection surfaces, and may lead to performance or compliance issues.

IV. Decision Pipeline for Function Inlining

Figure 1 illustrates the whole decision pipeline for function inlining, focusing on the LLVM [87] compiler infrastructure.

A. External Inlining Factors

1) (In)Direct Hints on Inlining at Source: The C and C++ programming languages offer (in)direct means for a programmer to hint function inlining using a directive or an attribute. As part of the C and C++ standards, the `inline` specifier provides a hint to inline a function at the function declaration [3]. Besides, different compilers provide additional attributes and pragmas with respect to inlining. Clang and GCC allow the compiler to explicitly inline the function with the specific function attribute of `__attribute__((always_inline))` regardless of the optimization level (*i.e.,* `-O0` is not an exception). Conversely, `__attribute__((noinline))` prevents the compiler from inlining the function [4]. In a similar vein, both `#pragma inline` and `#pragma noinline` directives serve the purpose of impacting function inlining. Note that the above compiler-specific directives are not part of the standards, which may differ depending on their implementations. Akin to direct hints above, other attributes can indirectly influence function inlining. The `__attribute__((flatten))` attribute allows for inlining all callees within a function as if each call site had `__attribute__((always_inline))`. Meanwhile, the `__attribute__((naked))` attribute is typically used for an assembly function, indirectly causing the compiler to avoid function inlining. Similarly, the

[3]The `__inline` specifier serves the same purpose before the C99 standard; however, it is still compatible with modern compilers.

[4]Similarly, `Microsoft Visual C++` compiler supports the `__forceinline` and `__declspec(noinline)` keywords to enforce and prevent function inlining, respectively.

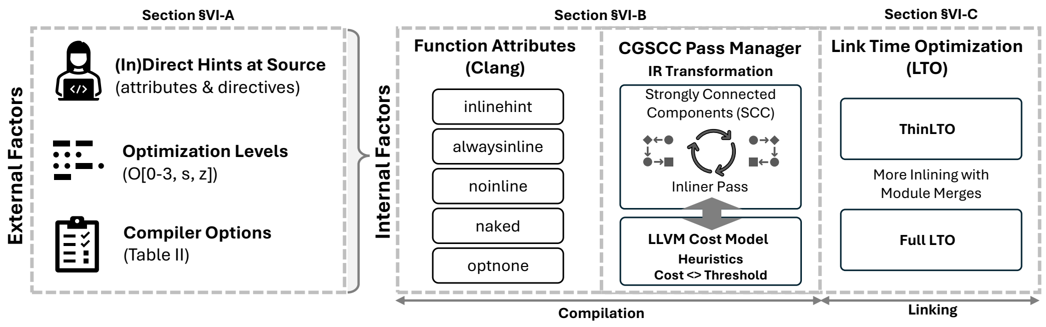


Fig. 1: Overview of the whole function inlining pipeline in the LLVM toolchain [87]. We simplify the pipeline of a complex decision process by classifying the factors that affect inlining into external and internal elements. Varying directives and attributes (*e.g.,* `inline` keyword) at source code, optimization levels, and compilation options allow a programmer to impact the inlining ratio in a binary (Section IV-A). At compilation time, Clang defines the internal attributes of a function based on those factors. Opt iteratively performs function inlining across strongly connected components (SCCs) during the transformations of intermediate representation (IR) (Section IV-B). Note that the process highly involves with the heuristics and the built-in cost model (*e.g.,* checking if a cost is higher than a threshold). In the case of link time optimization (LTO) enabled, additional function inlining can be performed by consolidating modules (Section IV-C).

TABLE II: Comprehensive compiler options that affect a cost model for function inlining at the frontend and middle end.

Option Name	Tool	Default	Description
`-finline-functions`	Clang	Disabled	Inlines a suitable function based on an optimization level
`-finline-hint-functions`	Clang	Disabled	Inlines a function that is (explicitly or implicitly) marked as `inline`
`-fno-inline-functions`	Clang	Disabled	Disables function inlining unless a function is declared with `always_inline`
`-fno-inline`	Clang	Disabled	Disables all function inlining except `always_inline`
`-inlinedefault-threshold`	Opt	225	Sets the initial threshold for O1 and O2 alone
`-inline-threshold`	Opt	225	Sets the initial threshold for all optimization levels
`-inlinehint-threshold`	Opt	335	Sets the threshold of a function marked with the `inlinehint` attribute
`-inline-cold-callsite-threshold`	Opt	45	Sets the threshold of a cold call site
`-inline-savings-multiplier`	Opt	8	Sets the multiplier for cycle savings during inlining
`-inline-size-allowance`	Opt	100	Sets the size allowance for inlining without sufficient cycle savings
`-inlinecold-threshold`	Opt	45	Sets the threshold for a cold call site
`-hot-callsite-threshold`	Opt	3,000	Sets the threshold for a hot call site
`-locally-hot-callsite-threshold`	Opt	525	Sets the threshold for a hot call site within a local scope
`-cold-callsite-rel-freq`	Opt	2	Sets a maximum block frequency for a call site to be cold (no profile information)
`-hot-callsite-rel-freq`	Opt	60	Sets a minimum block frequency for a call site to be hot (no profile information)
`-inline-call-penalty`	Opt	25	Sets a call penalty per function invocation
`-inline-enable-cost-benefit-analysis`	Opt	false	Enables the cost-benefit analysis for the inliner
`-inline-cost-full`	Opt	false	Enables to compute the full inline cost when the cost exceeds the threshold
`-inline-caller-superset-nobuiltin`	Opt	true	Enables inlining when a caller has a superset of the callee's `nobuiltin` attribute
`-disable-gep-const-evaluation`	Opt	false	Disables the evaluation of `GetElementPtr` with constant operands

`__attribute__((optnone))` attribute disables all optimizations for a function as well as inlining.

2) Optimization Levels: A compiler optimization level highly affects IR and machine code generation, striking a balance between performance, compilation time, and size. The `-O[0-3]` typically represents the level of optimization (*i.e.,* none, basic, medium, or maximum) whereas the `-O[s,z]` adjusts the code size (*i.e.,* small or minimum). The optimization strategy unavoidably impacts on function inlining: *e.g.,* `-O3` performs aggressive inlining to enhance execution speed, `-Os` does moderate inlining to balance performance and size, and `-Oz` attempts to minimize code size (*e.g.,* less inlining than others).

3) Compiler Options: LLVM [87] offers varying options that configure global settings for inlining decisions across all functions. Table II summarizes comprehensive (style-agnostic) options that assist in elaborately controlling the function inlining behaviors in LLVM [5]. For instance, `-finline-functions` allows the compiler to decide on a function to be inlined. Alternatively, `-finline-hint-functions` instructs the compiler to consider a function with a specific keyword or attribute for inlining alone. In contrast, Clang also provides the options to disable inlining based on compiler heuristics with `-fno-inline-functions` or to disable it entirely, except for a function marked `alwaysinline` (Section IV-B).

B. Internal Inlining Factors at Compilation

1) Function Attributes: Based on the external factors like hints, optimization levels, and compiler options, Clang internally labels the attribute of a function as one of the follow-

[5]The supportive options can be found with `--help-hidden`. The function-inlining-specific options available for the LLVM optimizer (opt) can be passed with `-mllvm [flagname=value]` in Clang. Note that we exclude the options that rely on a specific style or language (*e.g.,* GNU89, Assembly).

TABLE III: Selective call-site cases that never inline a function as the built-in heuristics in LLVM.

Scope	Case Description
Basic Block	A misuse of a block address outside of the specific instruction (*e.g.,* `callbr`)
Caller	`optnone` attribute that disables all optimizations including inlining `noduplicate` attribute that avoids multiple instances
Caller and Callee	Both callee and caller with conflicting attributes (*e.g.,* `alwaysinline` and `noinline` together) An incompatible null pointer definition between the caller and callee
Callee	`noinline` attribute An interposable function that can be replaced or overridden by another at link time or runtime A unsplit coroutine call that cannot split into separate parts for suspension and resumption
Call site	`noinline` attribute A `Byval` argument with an incorrect memory address space (*i.e.,* without the `alloca` address space) An indirect call where the function being called cannot be determined at compile time
Instruction	Amount of memory unknown at compile time (*e.g.,* dynamic allocation) A function with multiple return paths (returns twice) A function initialized with variadic arguments An indirect branch where the target is determined at runtime An intrinsic that is too complex to be inlined (*e.g.,* `icall_branch_funnel` or `localescape`). A recursive call that lead to infinite loops or allocate too much stack space

ing: `inlinehint`, `alwaysinline`, `noinline`, `naked`, and `optnone`. Setting those attributes occurs at the early stage of IR generation (from AST), conforming custom directives in a subsequent optimization process. For instance, although `-O0` disables most optimizations for compilation speed and debugging, it performs minimal and necessary transformations (*e.g.,* `alwaysinline`) to ensure correct code execution. As a final note, a compiler-generated function like an intrinsic function is often marked with `alwaysinline` to ensure efficiency.

2) LLVM Inliner Pass: LLVM offers the *inliner pass* that can analyze and transform intermediate representations, which internally maintains a pass manager for applying a chain of passes to IRs in a specific order [69]. The LLVM's function inliner pass performs the decision of function inlining, which is tightly coupled with the CGSCC pass manager [66] that runs on a strongly connected component (SCC) in a call graph. The SCC consists of callers and callees, where each function is simplified via a sequence of transformation passes such as control flow graph simplification (SimplifyCFG pass [70]), scalar replacement of aggregates (SROA pass [71]), and early common sub-expression elimination (EarlyCSE pass [67]). These preliminary steps ensure that non-trivial SCCs are adequately optimized prior to function inlining. Finally, CGSCC iteratively invokes the function inliner pass across SCCs in a bottom-up order based on a cost model (See Section IV-B3).

3) Cost Model: The cost model in LLVM begins with the pre-defined cases that must not be inlined (as heuristics). Table III enumerates such cases with a scope. Once the model confirms feasible cases for function inlining, the function inliner pass evaluates a call site (*i.e.,* a pair of caller and callee) to dynamically compute a *threshold* and a *cost* depending on the aforementioned factors. Simply put, the inliner pass mechanically performs inlining transformation in SCC when *the cost is less than the threshold.*

Threshold Computation. At each call site, a threshold is initialized by an optimization level (*e.g.,* `-Oz` $\rightarrow$ 5, `-Os` $\rightarrow$ 50, `-O1` $\rightarrow$ 225, `-O2` $\rightarrow$ 225, `-O3` $\rightarrow$ 250). Additionally, the CGSCC pass manager holds fruitful information on the call site with profile summary information (PSI; *e.g.,* hotness or coldness of a function, execution times) and block frequency information (BFI; *e.g.,* execution frequency of an individual basic block). Afterward, the inliner pass increases or decreases the threshold depending on ① the attributes of a caller and/or a callee, ② the property of a code region, and ③ other constructs like a complex branch or a vectorized instruction (Table IV). For instance, the `inlinehint` attribute via the `inline` specifier can increase the threshold.

Cost Computation. By default, a cost is initialized to zero. A special case arises when a callee is a static function and the call site is the last call invocation, setting the initial cost to

TABLE IV: Conditions that affect a cost and a threshold for function inlining for the LLVM's cost model.

Target	Impact	Condition
Cost	↑	Missing instruction simplification (*e.g.,* switch, loop, load) Presence of an intrinsic function Unoptimized call sites within a callee Complex memory operations that cannot be simplified Cold calling conventions (`coldcc`) General function call overhead (`call penalty`)
	↓	Last call of a static function Call site arguments with values (*e.g.,* `byval`) Successful transformation (*e.g.,* indirect to direct calls)
Threshold	↑	Presence of the `inlinehint` attribute Hot region Adjustments based on the target-specific architecture
	↓	Presence of the `minsize` or `optsize` attributes Cold region Complex branching Presence of vectorization instructions

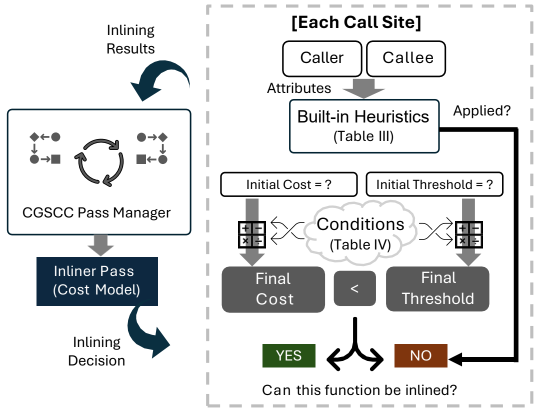


Fig. 2: Overview of the cost model in LLVM for determining a function to be inlined. While the CGSCC pass manager runs on strongly connected components, it invokes the inliner pass when needed. The cost model first considers the heuristics (Table III) of each call site to check if it can never be inlined. If possible, a cost and a threshold are initialized, followed by being updated according to dynamic conditions (Table IV). The process mechanically determines function inlining according to a cost. Then, CGSCC iteratively performs further optimizations based on the inlining.

a negative value of $-15,000$. This substantially increases the likelihood of function inlining. As summarized at Table IV, various conditions can adjust the cost. In essence, the inliner pass walks through each basic block and every instruction by adding or subtracting a designated value, obtaining the final cost. It is noteworthy to mention that a cost or a threshold could be configured by compiler options (Table II). On the one hand, the cost can be penalized (*i.e.,* increasing) in cases of missing instruction simplification, presence of an intrinsic function, an optimized call site, a complex memory operation, or a cold calling convention. On the other hand, the cost can decrease in cases of call site arguments with values and transformation from indirect to direct call invocation. LTO promotes every symbol to internal linkage using the `internalize` pass, attempting to eliminate it. However, the inlined function under LTO may remain if it has not been inlined at all call sites.

C. Internal Inlining Factors at Linking

1) Link Time Optimization and Inlining: Enabling LTO [39] literally provides the linker with global optimization opportunities from the whole program viewpoint at link time, which otherwise cannot be performed under the individual compilation unit. There are two primary types of LTO [6]. in LLVM: ThinLTO [47] and Full LTO [68], each with different implications for function inlining.

Inlining with Full LTO. Full LTO [68] enables aggressive inlining across the entire program by making all functions globally visible, allowing for further optimizations such as constant propagation and dead code elimination. However, it incurs significant resource overheads (*e.g.,* memory usage), taking a longer compilation time.

Inlining with ThinLTO. ThinLTO [47] aims to reduce computational overheads while adopting the benefits of Full LTO. In essence, ThinLTO generates a summary of the functions in each module, which allows for performing additional function inlining along with incremental cross-module information during the linking phase.

V. Subtleties in Function Inlining Practices

In this section, we clarify six (common) misleading beliefs about function inlining, none of which hold true in practice. Then, we introduce the extreme inlining strategy.

A. Misleading Case Study

A function would not be inlined with `-O0` or the `-fno-inline` option. One may want to generate a binary without any optimization (including inlining), thereby preserving a function and its inner structure eases debugging (*e.g.,* tracing an execution flow, identifying an error [10]). Although the compiler conforms with the request to suppress most optimizations, a function with the `always_inline` directive remains an exception regardless of a given optimization level or provided compiler option. In a nutshell, both `-O0` and the `-fno-inline` option cannot override the above directive, while it broadly prevents inlining as in Figure 8 and Figure 9.

[6]To enable LTO in LLVM, we use the `-flto` option using the LLVM linker (*i.e.,* `-fuse-ld=lld`).

The `always_inline` directive would always inline a function. Any function directive (including `inline` and `always_inline`) does not necessarily guarantee a function to be inlined because the compiler will make a final decision throughout the decision pipeline (Figure 1) to evaluate a call site with the complex cost model (Figure 2). There are plenty of such cases where the compiler never considers function inlining irrespective of optimization levels, directive hints, or compiler options, including recursive calls, indirect calls, and functions with variadic arguments (Table III).

A function without any inlining-relevant directive would not be inlined. In practice, LLVM regards every function as a candidate to be inlined during compilation, considering a function size, complexity, optimization level, and compiler options as a whole. In essence, function inlining is an opt-out operation in that a function may be inlined unless it has an explicit mark with the `no_inline` attribute Figure 7 and Figure 6 show such examples.

An inlined function would disappear in a binary. One of the most common misconceptions is that an inlined function has been eliminated (hence fully disappeared) because the call site is replaced with the function. The answer is partially affirmative; however, it is possible that an inlined function could be present (with a symbol) in a final binary when the function needs to be invoked globally Figure 7 and Figure 6 show such instances.

A conflicting condition for function inlining might lead to unpredictable behaviors. One may pose a hypothetical question like ① what if the two functions may have a contradictory condition (*e.g.,* one with `inline` and another with `noinline`)? or ② what if a conflicting option is given (*e.g.,* `-fno-inline`) when compiling a function with `inlinehint`)? To address such issues, the compiler establishes several pre-defined rules. First, the attributes (directives) come over the compiler options. Next, the compiler prioritizes each directive in the following order: `optnone`, `noinline`, `minsize`, `optsize`, `inlinehint`, and `always_inline`. To exemplify, when a function is declared with `inlinehint` or `always_inline`, `optnone` takes precedence, effectively negating other attributes. Similarly, the presence of `noinline` overrides `alwaysinline`.

A function in a library would not be inlined. It is common practice to resolve a function address in a library at runtime, internally following the routine via Procedure Linkage Table (PLT) and Global Offset Table (GOT), with the assistance of a dynamic loader. However, there are a handful of scenarios that a library function can be an inlining candidate. First, the preprocessor prepares the source by handing various directives (*e.g.,* macro definition, file inclusion, conditional statement), possibly rendering a function defined in a header file inlined when the header is included in multiple translation units. Second, similarly, it is possible for the compiler to inline a function if a header-only library [59], [52] provides complete function definitions visible in a header. Third, LLVM defines a series of special intrinsic functions [88] (*e.g.,* `llvm.memcpy`, `llvm.memset`) that operate on the IR level, being exposed as an inlining candidate with the `always_inline` attribute [7].

B. *Inlined but Remaining Functions*

A function symbol with an internal linkage (*e.g.,* `static`) goes away when inlined in every call site. However, even with an internal function, the function may remain after inlining (from elsewhere) if any call site requires that function invocation. Conversely, a function with an external linkage (*e.g.,* `extern`) would remain after being inlined by default.

C. *Extreme Inlining*

Distinguishing from aggressive inlining by high optimization levels, we introduce the concept of *extreme inlining* that promote the function inlining ratio beyond standard compiler behaviors. Section VI-B3 elaborates our exploration by exploiting a decision pipeline to establish a strategy for extreme inlining, including opt-levels, compilation options, and LTO.

VI. Function Inlining and Security Implications

We run our experiments on a 64-bit Ubuntu 20.04 system equipped with Intel(R) Xeon(R) Gold 5218R CPU 3.00GHz, 512GB RAM, and two RTX A6000 GPUs.

Benign Software Corpus. The first half of Table V summarizes our benign program corpus to evaluate the impact of function inlining on ML-based models. Each package and application may have been compiled with slightly different compilation options (*e.g.,* `-Oz`, `-O3`) or inlining thresholds (*e.g.,* 2,225, 200,000) due to build constraints. In cases where LTO (either thin or full) fails, we disable it but preserve the original inlining thresholds and optimization levels. To induce extreme inlining behavior, we empirically tune the relevant compilation settings. To minimize evaluation bias, we deduplicate identical functions by symbol names and ensure test samples never appear in training. Our final benign application dataset consists of $1,524$ program variations, including additional 398 samples with extreme inlining applied.

Malicious Software Corpus. The second half of Table V summarizes our malware corpus. We focus on IoT malware due to its prevalence, shared code bases [21], and the availability of leaked source code (*e.g.,* Mirai [9] and Gafgyt [78]). For detecting malware, we construct a curated dataset of 761 malware samples from VirusShare [77], including Mirai, Gafgyt, and Tsunami, by removing corrupted binaries, and packed executables from a broader collection. Similarly, for classifying malware, we prepare $12,727$ samples across 10 families from Alrawi *et al.* [7]: Mirai, Gafgyt, Tsunami, Lotoor, Dofloo, DDoSTF, ExploitScan, Dvmap, Gluper, and Healerbot. To evaluate the impact (*i.e.,* evading detection) of function inlining, we recompile open-sourced Mirai and Gafgyt (100 samples per each) by applying extreme inlining with custom compiler options. Our final malware dataset consists of $13,688$ samples in total. For malware-related tasks, we remove duplicate samples that are identical 62-dimensional statistical feature representation to avoid model overfitting. As a final note, Table XI in Appendix shows the whole recipe for extreme inlining in our experiments.

[7]The intrinsic functions in LLVM are beneficial, providing code simplification and platform independence as well as leveraging the features of a target architecture.

TABLE V: Binary corpus for experiments to assess the impact of function inlining on ML-based models. We choose various security tasks: (T1) binary code similarity detection, (T2) function name prediction, (T3) malware detection, (T4) malware family prediction, and (T5) vulnerability detection. We adopt two open-sourced malware for artificial inlining. Note that we generate 32-bit ARM binaries for T4(*), while the others are generated as 64-bit Intel x86 binaries.

Type	Package or Application(s)	Version	Baseline Binaries	T1	T2	T3	T4*	T5
Benign	**coreutils** [20]	9.3	106	742	742	624	-	-
	binutils [13]	2.40	22	118	118	78	-	-
	diffutils [26]	3.8	4	28	28	24	-	-
	findutils [35]	4.9	4	28	28	24	-	-
	openssl [81]	3.1.4	1	7	7	6	-	-
		1.0.2d	1	-	-	-	-	6
	lvm2 [76]	2.03.21	52	364	364	-	-	-
	gsl [40]	2.7.0	2	14	14	9	-	-
	valgrind [99]	3.21.0	4	28	28	156	-	-
	openmpi [79]	4.1.5	7	49	49	36	-	-
	putty [89]	0.79	6	42	42	36	-	-
	nginx [80]	1.21.6	1	7	7	6	-	-
	lighttpd [62]	2.0.0	2	14	14	10	-	-
	SPEC2006 [97]	-	15	-	-	83	-	-
Malware	**Mirai** [37]	-	2	-	-	261	9,215	-
	Gafgyt [27]	-	1	-	-	421	3,119	-
	Tsunami	-	-	-	-	79	228	-
	Lotoor	-	-	-	-	-	80	-
	Dofloo	-	-	-	-	-	40	-
	DDoSTF	-	-	-	-	-	15	-
	ExploitScan	-	-	-	-	-	11	-
	Dvmap	-	-	-	-	-	8	-
	Gluper	-	-	-	-	-	6	-
	Healerbot	-	-	-	-	-	5	-

A. *Security Impact of Function Inlining on ML Models*

Research Questions. We define five research questions to evaluate how function inlining affects ML-based models from a security aspect. We choose five ML-based security tasks, including binary code similarity detection (T1), function name prediction (T2), malware detection (T3), malware family classification (T4), and vulnerability detection (T5).

- **RQ1**: How does function inlining affect ML-based binary code similarity detection models (Section VI-A1)?
- **RQ2**: How does function inlining affect ML-based function symbol name prediction models (Section VI-A2)?
- **RQ3**: How does function inlining affect ML-based malware detection models (Section VI-A3)?
- **RQ4**: How does function inlining affect ML-based malware family classification models (Section VI-A4)?
- **RQ5**: How does function inlining affect an ML-based vulnerability detection model (Section VI-A5)?

ML Model Selection for Security Tasks. We select a diverse set of ML-based models spanning both discriminative and generative tasks, covering a range of architectures (*e.g.,* embedding-based models, GNNs, Transformers, RNNs) and input modalities (*e.g.,* statistical features, dynamic traces). Note that we use the original pre-trained form for T1 and T5. Meanwhile, we retrained an AsmDepictor model for T2 to match our Clang-based corpus, as their originals were trained on GCC binaries. Similarly, SymLM was retrained on our dataset so that execution-aware embeddings can reflect the semantics of our compilation settings. For T3 and T4, we reimplemented the model architectures based on the descriptions in the original paper [2], and retrained the models on our corpus.

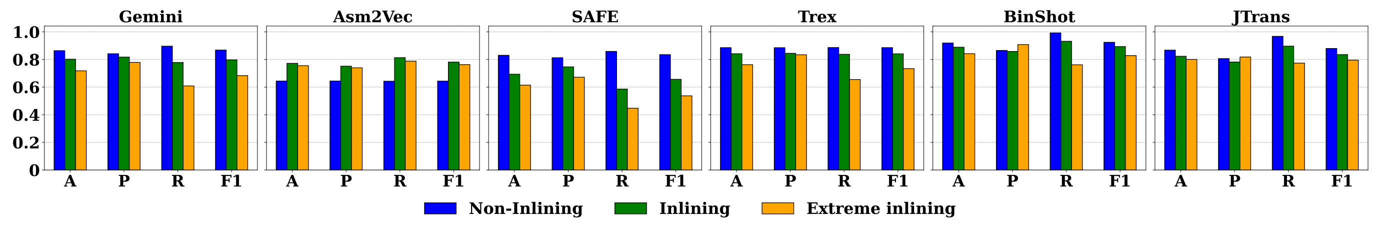


Fig. 3: Experimental results across six BCSD models on three datasets: non-inlining, inlining, and extreme inlining pair samples. We carefully prepare a dataset comprising non-inlining, inlining, and extreme inlining cases to investigate the effectiveness of inlining. Overall, there is a gradual decrease in performance for the inlining samples. We discuss the exceptional case of Asm2Vec in Section VI-A1. In general, the other models demonstrate a significant drop in recall compared to precision (*i.e.,* increasing false negatives). A, P, R, and F1 denote an accuracy, precision, recall, and F1 value, respectively.

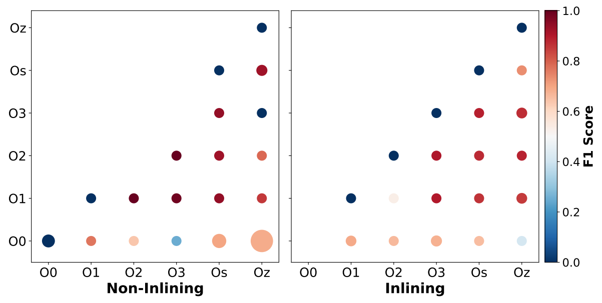


Fig. 4: Asm2Vec performance for the non-inlining (left) and inlining (right) pairs. Circle area indicates the number of pairs, and color intensity represents their F1 score. The breakdown shows that most performance degradation stems from non-inlining pairs involving `-O0` versus other optimization levels, suggesting that random walk sequences at `-O0` differ substantially from those at higher optimization levels. Empty cells correspond to combinations with too few samples.

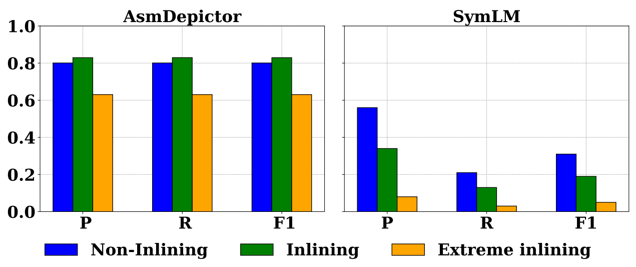


Fig. 5: Experimental results with AsmDepictor [55] and SymLM [46] on three datasets: non-inlining, inlining, and extreme inlining (Section VI-A2). AsmDepictor shows a slight performance increase with the inlining cases, demonstrating that it can leverage the additional contextual information for inference. SymLM demonstrates a decrease in performance for the inlining case. Meanwhile, the extreme inlining cases mislead both models with unseen patterns, resulting in non-negligible performance drops (*e.g.,* 23.99% for AsmDepictor, 75.69% for SymLM).

1) (RQ1) Inlining Impact on BCSD Models: A BCSD task determines code similarity between a given pair of code snippets. To examine the function inlining impact on performance, we prepare three different pairs: non-inlining samples with the pairs of both non-inlining functions, inlining samples with the pairs of a non-inlining and an inlining function, and extreme inlining samples with the pairs of a non-inlining and an inlining function that applies our extreme inlining strategy We use the functions that appear in every optimization, highlighting the effectiveness of inlining on various BCSD models. We choose six representative BCSD models for evaluation: Asm2Vec [28], BinShot [5], Gemini [104], Trex [85], JTrans [100], and SAFE [75]. Finally, we prepare $14,984$, $95,774$, and $23,378$ function pairs from BinShot [5], JTrans [100], and the BCSD benchmark [74], respectively.

Results. Figure 3 presents the performance comparison for six BCSD models with pairs of non-inlining, inlining, extreme inlining. We adopt the BCSD benchmark [74] that includes Asm2Vec [28], Gemini [104], Trex [85], and SAFE [75], providing a similarity score (S) between pairs in a single platform. We measure the accuracy, precision, recall, and F1 with a threshold of $T = 0.5$ (*e.g.,* similar if $S > 0.5$) in comparison with BinShot [5]. In general, moderate performance degradation has been observed (3.6% in F1; 6.2% of recall on average) for the inlining samples while significant decline (12.6% in F1; 21.2% in recall on average) for the samples with extreme inlining. Particularly, we observe notable decrease in recall (false negatives) rather than precision (false positives). While this suggests that similar function pairs may appear dissimilar when evaluating inlined functions, one could argue that the additional semantic information may help the model better recognize the similarity between functions. However, as discussed in Section IV, function inlining introduces opportunities for further optimizations, potentially obscuring the similarity between otherwise a similar function pair during evaluation. Exceptionally, Asm2Vec deviates from other models: *i.e.,* non-inlining pair samples exhibits lower performance than inlining ones. Our investigation reveals that pair-wise performance discrepancies arises from the feature of statically tracing (*i.e.,* potential execution paths) by random walk to infer code similarity as illustrated in Figure 4. We observe that the pairs of `-O0` versus `-Oz` that account for almost half (10,614 out of 23,378) lead to overall performance degradation. We attribute this effect to differences in the random-walk sequences generated under `-O0` compared to those under higher optimization levels.

2) (RQ2) Inlining Impact on Function Name Prediction Models: A function name prediction task aims to infer the original symbol name of a function given a chunk of assembly code. Similar to the BCSD experiments, we prepare our dataset (Table V) for function name inference. We choose two representative models: AsmDepictor [55] (generative) and SymLM [46] (discriminative). We use precision (P), recall (R), and F1 for evaluation.

Results. Figure 5 presents the empirical results for AsmDepictor [55] and SymLM [46] on our test dataset (*e.g.,* 7,863 cases). AsmDepictor demonstrates improved performance with inlining, effectively leveraging the additional contextual information provided. While curious readers may attribute AsmDepictor's improved inlining performance to training data distribution where 79% of our training dataset consists of non-inlining functions. However, both AsmDepictor and SymLM experience significant performance degradation under extreme inlining: *e.g.,* 24.0% and 75.7%, respectively. As a generative model, AsmDepictor struggles with unseen instances when generating function symbol names, likely due to abnormal inlining patterns. The underperformance of SymLM may stem from its approach of treating function name inference as a multi-class multi-label classification task. Because the original work uses four architectures and incorporates obfuscation techniques for training, our findings suggest that achieving comparable performance requires a substantially larger dataset than what was used in our evaluation.

3) (RQ3) Inlining Impact on Malware Detection Models: A malware detection task aims to distinguish malware from benign sample(s). We trained four traditional ML models: logistic regression as a linear model, random forest as an ensemble method, CatBoost as a gradient boosting, and K-nearest neighbors (KNN) as instance-based learning; and two deep learning models: deep neural network (DNN) and convolutional neural network (CNN) proposed by Abusnaina *et al.* [2]. We extract 62 statistical semantic features with TikNib [53]. Then, we evaluate each model 10 times using Monte Carlo cross-validation.

Results. Table VI presents the experimental results for malware detection. While the original models achieved high accuracies, their performance has been moderately degraded (*e.g.,* around 20%) when tested on in-house malware samples that are applied with extreme inlining. This decline illustrates how inlining introduces substantial changes in code structure, leading to misclassification.

4) (RQ4) Inlining Impact on Malware Family Prediction Models: A malware family prediction task aims to forecast the likelihood that an unseen malware mutation belongs to a known family. Close to malware detection, we use the same ML-centric approaches excluding benign applications. It is noted that these malware targets the (prevalent) 32-bit ARM architecture in IoT devices. To handle a class imbalance, we select 500 samples per family with oversampling and down sampling [95].

Results. Table VI demonstrates the experimental results for malware family prediction (*e.g.,* 10 families). Using the same semantic features with malware detection experiments, all ML models show a decent performance (*e.g.,* around 87%) on community-collected IoT malware. However, their performance significantly declines: *i.e.,* 40% $\downarrow$ for most malware family prediction models when tested on our in-house malware with extreme inlining. Our findings indicate that all models are sensitive to our extreme inlining. We hypothesize that this senstivity arises from code sharing among IoT malware families [21], where inlining obscures structural patterns essential for fine-grained classification. In contrast, the broader semantic gap in malware detection appears to be less affected.

5) (RQ5) Inlining Impact on Vulnerability Detection Models: A vulnerability detection task identifies the presence of a known vulnerability (binary classification). Excluding vulnerability detection models that operate on source code [42], [63], [61], [58], [102], [101], we choose two binary-based detection models [73], [74] that adopt a code search (*i.e.,* probing a vulnerable function) by leveraging BCSD capability. We select 12 vulnerable functions in the libcrypto library from two firmware images [16] – four functions from Netgear R7000 and eight functions from TP-Link Deco M4 [74] (Table IX in Appendix). Accordingly, we compile two versions of the library containing the aforementioned vulnerabilities in OpenSSL-1.0.2d [81]: one with inlining enabled and one without. Each version is compiled at six optimization levels (`-O[0-3,s,z]`), resulting in $12,958$ and $13,298$ functions, respectively. We rank those functions based on their similarity to each known vulnerable function and then compute the mean reciprocal rank (MRR). We adopt MRR@100, which evaluates the Top 100 most similar functions.

Results. Table VII presents the MRR@100 scores before and after inlining, demonstrating the negative impact (*i.e.,* lower scores) on recognizing a vulnerability with an average drop of 69%. Notably, we observe an exception with the Trex model [85] at Netgear R7000 (0.550 $\rightarrow$ 0.625). However, our further analysis reveals that the ranks of other similar functions decrease except for the highest rank.

B. Evaluation of Function Inlining

Research Questions. This section explores the impact of function inlining transformation itself in terms of inlining ratios and static features, with the following research questions.

- **RQ6**: How do optimization levels affect a function inlining ratio across different applications (Section VI-B1)?
- **RQ7**: How do the default compiler options (provided by clang) affect a function inlining ratio (Section VI-B2)?
- **RQ8**: Which combination of compiler options (provided by opt) increase a function inlining ratio toward extreme inlining in practice (Section VI-B3)?
- **RQ9**: To what extent are static features affected by function inlining in an executable, such as instructions, control flow graphs, and call graphs (Section VI-B4)?

1) (RQ6) Function Inlining Ratio: Figure 6 illustrates the overall flow of inlined functions from the coreutils binaries compiled with `-O[0,3]` and our extreme inlining strategy (Section VI-B3). Starting from the whole $2,070$ functions (assuming little inlining with `-O0`), we observe more than half inlined functions ($1,183$). Out of those inlined functions, 997 were eliminated, while others remained. Driving to extreme inlining (more aggressive than `-O3`), approximately two-thirds are removed.

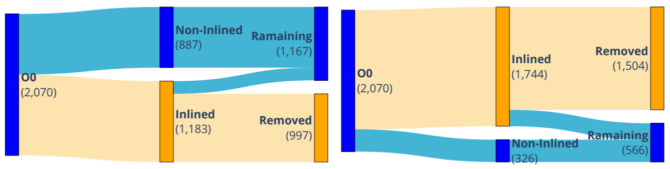
Function Inlining across Optimization Levels and Applications. Figure 7 presents the function inlining ratio across eight packages in a baseline dataset with five optimization

TABLE VI: Performance comparison of four classical ML models (*e.g.,* logistic regression, random forest, CatBoost, and k-Nearest Neighbors (KNN)) and two deep-learning-based models (*e.g.,* a deep neural network and a convolutional neural network) proposed by Abusnaina *et al.* [2] for malware detection (Section VI-A3) and malware family classification (Section VI-A4) tasks. Note that we generate malware variants with extreme inlining where their sources have been leaked (Mirai [37] and Gafgyt [27]). For both tasks, the performance of each model significantly drops due to the static feature perturbations (*e.g.,* call graphs, control flow graphs): around 20% ↓ for malware detection and approximately 40% ↓ for malware family prediction.

Security Task	**ML Model**	**Malware in the Wild**				**Malware with Extreme Inlining**			
		Accuracy	**Precision**	**Recall**	**F1**	**Accuracy**	**Precision**	**Recall**	**F1**
Malware Detection	Logistic Regression	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.81 ± 0.03**	**0.84 ± 0.02**	0.81 ± 0.03	**0.81 ± 0.03**
	Random Forest	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	0.75 ± 0.13	0.77 ± 0.13	0.75 ± 0.13	0.74 ± 0.14
	CatBoost	0.98 ± 0.02	0.98 ± 0.02	0.98 ± 0.02	0.98 ± 0.02	0.77 ± 0.01	0.84 ± 0.00	0.77 ± 0.01	0.75 ± 0.01
	KNN	0.93 ± 0.01	0.93 ± 0.01	0.93 ± 0.01	0.93 ± 0.01	0.78 ± 0.00	0.80 ± 0.01	0.78 ± 0.00	0.77 ± 0.00
	Abusnaina *et al.* [2] (CNN)	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	0.78 ± 0.01	0.75 ± 0.02	**0.83 ± 0.03**	0.79 ± 0.01
	Abusnaina *et al.* [2] (DNN)	**0.99 ± 0.01**	**0.99 ± 0.01**	0.99 ± 0.00	**0.99 ± 0.01**	0.75 ± 0.02	0.72 ± 0.01	0.82 ± 0.05	0.76 ± 0.02
	Average	0.98 ± 0.01	0.98 ± 0.01	0.98 ± 0.01	0.98 ± 0.01	0.77 ± 0.05	0.78 ± 0.04	0.79 ± 0.01	0.78 ± 0.05
Malware Family Prediction	Logistic Regression	0.88 ± 0.05	0.90 ± 0.06	0.88 ± 0.05	0.88 ± 0.06	0.48 ± 0.06	0.45 ± 0.07	0.48 ± 0.06	0.46 ± 0.05
	Random Forest	**0.91 ± 0.05**	**0.92 ± 0.05**	**0.91 ± 0.05**	**0.90 ± 0.06**	0.50 ± 0.00	0.47 ± 0.06	0.50 ± 0.00	0.48 ± 0.04
	CatBoost	0.89 ± 0.03	0.90 ± 0.03	0.89 ± 0.03	0.88 ± 0.03	**0.53 ± 0.10**	**0.62 ± 0.30**	**0.53 ± 0.10**	**0.54 ± 0.17**
	KNN	0.90 ± 0.06	0.92 ± 0.06	0.90 ± 0.06	0.90 ± 0.07	0.44 ± 0.05	0.49 ± 0.03	0.44 ± 0.05	0.46 ± 0.04
	Abusnaina *et al.* [2] (CNN)	0.82 ± 0.06	0.85 ± 0.05	0.82 ± 0.06	0.81 ± 0.06	0.23 ± 0.18	0.43 ± 0.13	0.23 ± 0.18	0.25 ± 0.14
	Abusnaina *et al.* [2] (DNN)	0.78 ± 0.07	0.80 ± 0.07	0.78 ± 0.07	0.76 ± 0.09	0.36 ± 0.15	0.29 ± 0.15	0.36 ± 0.15	0.31 ± 0.13
	Average	0.87 ± 0.05	0.88 ± 0.05	0.87 ± 0.05	0.87 ± 0.05	0.44 ± 0.09	0.46 ± 0.09	0.43 ± 0.09	0.41 ± 0.10

TABLE VII: Performance comparison (in MRR@100) of four vulnerability detection models with function inlining. We select 12 vulnerable (target) functions in libcrypto from two firmware images [16] (Netgear R700 and TP-Link Deco-M4). Then, we prepare a list of comparable functions in OpenSSL-1.0.2d [81] with and without inlining for probing the vulnerable ones. The performance of each model mostly significantly drops after inlining has been applied. We compute the mean reciprocal rank for evaluation. (Section VI-A5).

Model	**Netgear R7000**		**TP-Link Deco-M4**	
	Non-inlining	**Inlining**	**Non-inlining**	**Inlining**
Gemini [104]	0.049	0.000	0.321	0.004
Asm2Vec [28]	0.256	0.128	0.016	0.008
SAFE [75]	0.125	0.003	0.018	0.003
Trex [85]	0.550	0.625	0.286	0.048



(a) High optimization (`-O3`) (b) Extreme inlining

Fig. 6: Visualization of inlined functions and their presence in the coreutils package when compiled with (a) the `-O3` optimization level and (b) our extreme inlining strategy. Compared to `-O0`, more than half of the functions have been inlined and yet eliminated in `-O3` (Section VI-B1).

levels excluding `-O0`. First, for all optimization levels, coreutils displays the highest inlining ratio whereas nginx ranks the lowest in most optimizations. Second, in every package, `-Os` and `-Oz` consistently show lower inline ratios because of the optimization purpose to reduce a binary size (*i.e.,* the inline operation of a function increases a size). Third, a function inlining ratio can vary depending on applications. Lastly, a non-negligible number of inlined functions has remained (predominately) due to the function with an external linkage or even an internal linkage that has not been inlined at all call sites. Figure 8 illustrates the cumulative distribution of the inlining ratio across all binaries in our baseline dataset. Notably, a slight function inlining ratio has been observed even with `-O0` (*e.g.,* `always-inline`) shows a maximum ratio of 9.52% and a mean of 0.83% In contrast, `-O[1-3]` collectively demonstrate a significantly high inlining ratio with a maximum of 66.67% and a mean of 32.89%, 32.77%, and 32.70%, respectively. We observe similar maximum inlining ratios of 65.83% for `-Os` and 61.67% for `-Oz`. However, the mean inlining ratios slightly decrease to 27.42% for `-Os` and 20.47% for `-Oz`, indicating those optimization tactics stick to lowering a binary size.

2) (RQ7) Compiler Options (provided by Clang) Impact on Function Inlining ratio: Figure 9 depicts the impact of enabling four default compiler options (`-finline-hint-functions`, `-fno-inline-functions`, `-finline-functions`, `-fno-inline`) relevant to function inlining while the default means no such options are given. The ratio with the `-finline-functions` option is identical to that with the default setting. Meanwhile, `-finline-hint-functions` shows a relatively low inlining rate. In SPEC2006, the `-fno-inline` and `-fno-inline-functions` options are effective. It is noted that `always_inline` contributes to maintain a small number of function inlining in coreutils even with the inline-suppressing options.

3) (RQ8) Compiler Options (provided by Opt) Exploration toward Extreme Inlining: We investigate 12 compiler options associated with function inlining, aiming to seek a combination of the compiler options toward extreme inlining according to the cost model in LLVM. Motivated by BinTuner [92], we started by setting a certain value that contributes to the inlining ratio. However, we observe that adjusting the initial value could significantly increase compilation time. To examine the inlining ratio within a reasonable compilation time, we

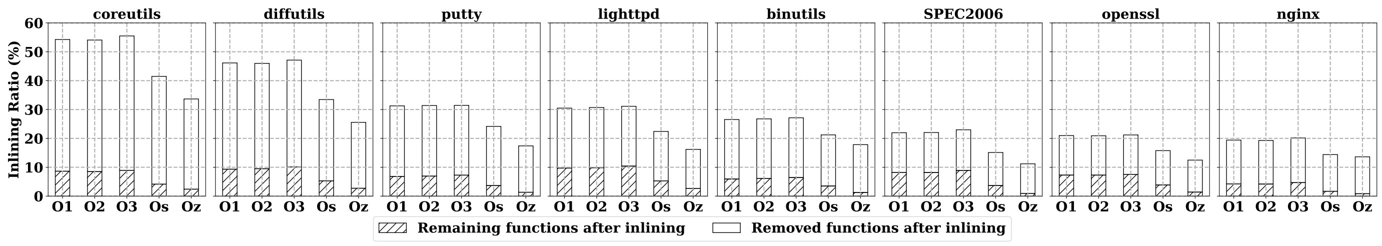


Fig. 7: Comparison of function inlining ratios compiled with different optimization levels (`-O[0-3,s,z]`) across eight applications. As the inlining decision of an individual function largely relies on the dynamics of a cost and a threshold by the cost model (Section VI-B1), inlining ratios may considerably differ. For example, coreutils demonstrates the highest (*e.g.,* more than half) in all optimization levels, whereas nginx ranks the lowest. The ratios in `-Os` and `-Oz` are relatively smaller than `-O[0-3]`. While most inlined functions were eliminated, a non-negligible number (around 10%) remained.

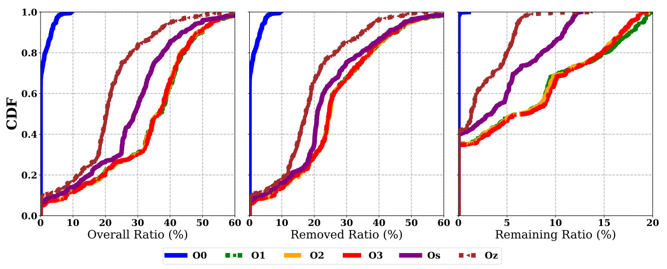


Fig. 8: Cumulative distribution function (CDF) of function inlining ratios across all binaries in our dataset compiled with various optimization levels `-O[0-3,s,z]` (Section VI-B1). An inlining optimization occurs even with `-O0` while an aggressive optimization (`-O3`) drives more inlining.

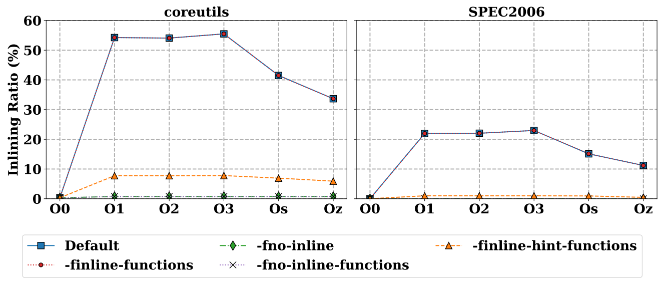


Fig. 9: Effectiveness of compiler-provided options that globally affect the behavior of inlining (Section VI-B2). We measure function inlining ratios on coreutils and SPEC2006 across various optimization levels of `-O[0-3,s,z]`. We confirm the `-fno-inline-functions` and `-fno-inline` options effectively prohibit an inlining behavior.

increment a specific value by a fixed interval, such as 500 (*e.g.,* [0, 500, 1,000, ...]) for benign applications and 50,000 (*e.g.,* [0, 50,000, 100,000, ...]) for malware. For an option defined as a boolean value, we flip the default value (*e.g.,* $true \rightarrow false$).

Results. Compared to other options that show marginal differences in a function inlining ratio, it tends to be proportional to `-inline-threshold` and inversely proportional to `-inline-call-penalty` (Figure 10). The `-inline-threshold` option overrides the global threshold value while the `-inline-call-penalty` option adjusts the call penalty value to calculate a cost. For example, the inlining ratio increases when a threshold increases from the default value (*e.g.,* $225 \rightarrow 2,225$). Notably, these compiler options do not bring about the heuristics at Table III.

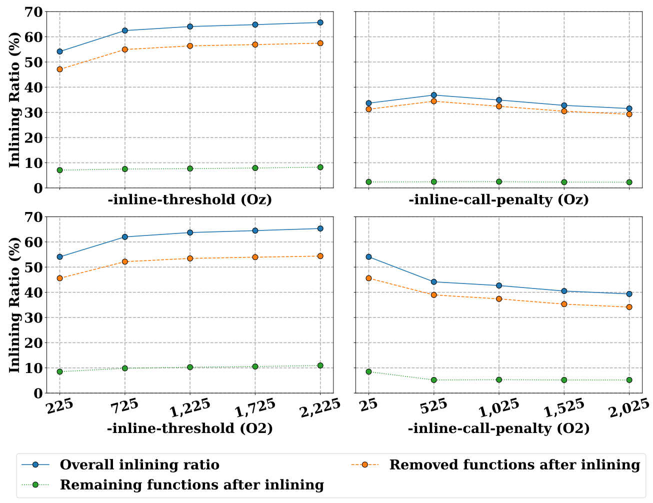


Fig. 10: Selective results of our exploration for seeking a combination of compiler options. With coreutils, we extensively tune (*i.e.,* increase or decrease) the values of the compiler options (Table II), which can eventually affect function inlining. Our findings show that heuristically increasing `-inline-threshold` and decreasing `-inline-call-penalty` assist in growing an overall function inlining ratio (Section VI-B3).

Function Inlining and LTO. Next, we investigate how LTO affects overall function inlining ratios. LTO enables an additional round of optimization after the initial pre-link optimization on each object file. We examined four scenarios: no LTO, Full LTO, ThinLTO, and LTO with our extreme function inlining. We set the `-inline-threshold` option to 200,000 with Full LTO, successfully compiling three packages (coreutils, diffutils, and findutils). As in Figure 11, Full LTO exhibits a higher inlining ratio than ThinLTO because Full LTO operates on a single thread with all object files available, whereas ThinLTO uses multiple threads and summary information. Interestingly, ThinLTO shows a higher inlining ratio than Full LTO at `-O1`, potentially due to more efficient, aggressive optimizations. LTO also leverages the internalize pass, converting external to internal linkage at link time, which increases the inlining ratio [47]. Our extreme inlining strategy

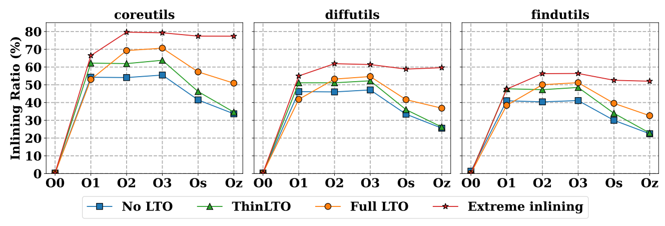


Fig. 11: Effectiveness of link time optimization (LTO) across different optimization levels and applications. Enabling LTO clearly demonstrates a higher function inlining ratio at `-O1` or above optimization levels. It is worthwhile to note that our extreme function inlining strategy even surpasses the ratio with Full LTO (*e.g.,* 79.64% in coreutils).

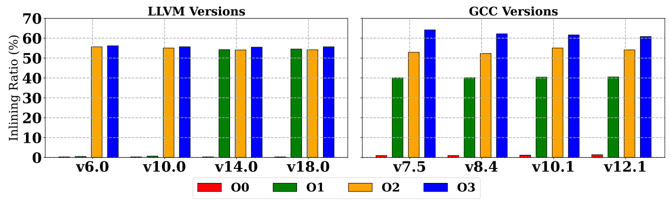


Fig. 12: Comparison of function inlining ratios across different optimization levels (O0–O3) for multiple versions of LLVM (left) and GCC (right). Higher optimization levels consistently yield higher inlining ratios across all compiler versions, with GCC exhibiting more variability between versions compared to LLVM.

achieves an extremely high inlining ratio, reaching 79.64% in coreutils.

4) (RQ9) Static Features Analysis According to Function Inlining: We analyzed 62 statistical features pertaining to instructions, CFG, and CG using TikNib [53]. For brevity, we select 18 features with the highest median gap (Figure 14 in Appendix). Note that we display the whole static feature names at Table X in Appendix. For a fair comparison, we normalize the data and remove the outliers with the three-sigma rule [51], which excludes any data points that fall beyond three standard deviations from the mean. We compared `-O0` against extreme inlining across 106 binaries in coreutils. Figure 14 in Appendix shows that extreme inlining drastically changes the statistical features of an executable binary. For example, there is a significant increase in arithmetic instructions (Features 6 and 8), likely due to the inlining of functions with arithmetic operations across multiple call sites. Additionally, we observe a decrease in the number of loops (Features 38 and 39) but an increase in loop size (Features 46, 48, and 49). This suggests that inlining results in fewer but larger loops. Extreme inlining may enable more aggressive optimizations, such as loop unrolling and removing smaller loops. These changes significantly impact the statistical features, which we expect will affect ML models relying on such features.

VII. Implementation

Ground Truth Extraction for Executable Binaries. We compile varying software in the ELF (Executable and Linkable Format) format with the `-g` option, which generates debugging information. We develop a script in Python with the `pyelftools` library [12] to parse ELF and extract the `DW_AT_inline` attribute (*i.e.,* inlined function) in a function symbol from DWARF [19]. Identifying a list of callers with the `DW_TAG_subprogram` tag at each function, we traverse child nodes with the `DW_TAG_inlined_subroutine` tag in each DWARF entry, which indicates that a function contains multiple inlined functions. Besides, we leverage TikNib [54] to extract 62 features (Table X in Appendix) that can impact a control flow graph, a call graph, and instruction sequences.

ML-based Models. First, we leverage the BCSD benchmark implementations [74] to assess a BCSD task instead of reinventing the wheel, including Asm2Vec [28], Gemini [104], Trex [85], and SAFE [75]. We adopt the original implementation of BinShot [5], which is unavailable on the above benchmark. Second, for malware detection and family prediction, we follow the Lei *et al.* [60]'s approach to generate four traditional ML models (Table VI) with the Scikit-learn library [25]. Additionally, we re-implement two deep learning models proposed by Abusnaina *et al.* [3] with TensorFlow [1] (Table VI). Third, we retrained AsmDepictor [55] and SymLM [46] with our dataset compiled in LLVM for function name prediction. Fourth, we assess vulnerability detection by leveraging the BCSD benchmark implementation [74] and available vulnerable firmware corpus [16].

VIII. Discussion and Future Work

Unreported Function Inlining Cases in DWARF. Although DWARF holds plentiful information about function inlining, there are several cases that (deliberately) do not report them. We observe missing the record of inlining when the whole inlined function body has been removed (*e.g.,* dead code elimination) during transformations. Another unreported case is when a specific intrinsic function has been inlined into other functions. This may cause a discrepancy between the output of the function inliner pass (*e.g.,* `-Rpass=inline`) and that of parsing DWARF.

Dataset Representativeness. Although we carefully select a wide range of different applications as a dataset for our experiments, it may not sufficiently represent all binaries to safely generalize our findings. We leave more scenarios and edge cases as our future work, which need to be explored with a diverse dataset to unearth the behavior of inlining a function.

Behavioral Differences between Compilers. Empirically, it is possible to have inconsistent outcomes due to the dynamics of function-inlining for code optimization across compilers and compiler versions. Figure 12 presents the function inlining ratios across different versions of GCC and LLVM compiler at various optimization levels. While LLVM exhibits relatively consistent inlining behaviors, GCC shows more aggressive inlining, particularly at higher optimization levels. Although our experiments focus on a specific version of LLVM, our observation remains applicable across its versions, providing valuable insights into how modern compilers handle function inlining. A deeper investigation into GCC's internal inlining decisions is left for future work.

Evaluation Metric for Function Inlining Ratio. We utilize the evaluation metric with the ratio of function inlining based on the number of (inlined) functions. However, an inlining decision is made at the call site granularity that takes a caller and a callee into account. Hence, the case of a single function that has been inlined into multiple locations or a chain of (nested) inlining may not been represented with our metric.

Potential Mitigations. We propose several potential mitigations: ① augmenting model training with inlined data, ② employing adversarial training that incorporates compiler-aware transformations, and ③ applying inlining-aware pre-processing (e.g., Highliner [22] uses de-inlining heuristics). We direct interested readers to Appendix Table XII for preliminary results on augmented training with inlined functions.

IX. Related Work

Feature Engineering for Binary Reversing. Genius [34] utilizes an attributed CFG (ACFG) where they incorporate eight features (statistical and structural) at the basic block granularity. Similarly, DL-FHMC [2] introduces 23 additional features on top of CFG. Additionally, ImOpt [45] explores features at the IR level to tackle compiler optimization and obfuscation techniques. In contrast, αDiff [65] utilizes a deep neural network to directly extract features. TikNib [53] extracts 72 statistical features using a large-scale benchmark, which demonstrates that simple interpretable models show comparable performance to state-of-the-art deep learning models with precise features. Note that we borrow 62 applicable features at the binary level to investigate the impact of inlining.

Function Inlining and Binary. Several prior works deal with function inlining from engineering aspects. Damásio *et al.* [23] focus on inlining for code size reduction, while Theodoridis *et al.* [98] explore optimal inlining strategies. In parallel, other studies examine function inlining in the context of binary analysis. One close effort to our work is Jia *et al.* [43], who investigate the impact of function inlining on binary similarity analysis. Bingo [14] introduces a dynamic inlining-simulation strategy that recursively expands callee functions to improve similarity detection. Similarly, Asm2Vec [28] adapts the Bingo's strategy for static analysis using selective callee expansion. FSmell [64] proposes a ML–based framework to detect inlined functions through instruction topology graphs. Meanwhile, Koo *et al.* [57] and AsmDepictor [55] emphasize the importance of a deeper understanding of function inlining. Nonetheless, many ML-based studies [86], [32], [103], [33], [105], [46], [83], [49], [84], [11] for binary analysis tend to underestimate or overlook inlining effects without thorough investigation. Unlike previous approaches, our work demystifies the compiler's inline decision process and the possibility of misuse by deliberately crafting evasive binary mutations through extreme inlining.

ML-assisted Approaches for Static Binary Analysis. Over the past decade, the widespread adoption of ML-assisted security tasks across various fields have demonstrated promising results. Such examples include malware detection, malware family classification, BCSD, and function name prediction tasks. In the area of BCSD, VulSeeker [38] employs a Siamese network-based graph embedding model to enhance similarity detection. InnerEye [110] and SAFE [75] adopt natural language processing (NLP) approaches for learning semantics from assembly code. In a similar vein, Asm2Vec [28] and DeepBinDiff [30] utilize unsupervised learning for training in the context of instructions, thereby improving their ability to detect code similarities. Lately, BinShot [5] learns a weighted distance vector with to better take dissimilar codes apart, which we include for our evaluation. For malware applications, Alasmary *et al.* [6] conduct a comprehensive CFG-based IoT malware detection task on Android. Cozzi *et al.* [21] examine the lineage of malware families, tracking their relationships across variants. Furthermore, BinTuner [92] and CARDINAL [48] attempt to re-compile malware samples with different compilation option settings to defeat signature-based approaches that rely on static engineering features. Meanwhile, recent advances in an attempt to recover lost information during compilation presents the inference of function and variable names, types, and even decompiled code. Debin [41] focuses on recovering variable names and types that learn the underlying semantics from instruction sequences at the binary level. NERO [24] trains enriching representations of call sites based on augmented control flow graphs to predict function symbol names. Similarly, AsmDepictor [55] leverages the state-of-the-art Transformer-based model into a function name prediction task, which is included in our evaluation.

Study on Compiler Flags and Behaviors. Dong *et al.* [29] and Zhang [109] *et al.* study symbolic execution to explore compiler flags: the former compare different optimization levels, while the latter empirically analyze GCC/Clang optimization flags. Ren *et al.* [93] focus on LLVM peephole optimizations for binary diffing, and BinTuner [92] explores large compiler flag spaces for binary similarity. Meanwhile, our work targets ML-based binary analysis and moves toward extreme inlining to evaluate and evade ML models by systematically examining LLVM's inlining code base.

X. Conclusion

Function inlining is a well-known optimization technique by modern compilers. Such an inlining behavior can considerably affect static features for a binary reversing task; however, it has yet well explored despite its importance and impact. In this work, we first conduct a comprehensive study focusing on function inlining, including (but not limited to) the investigation of its decision pipeline, compiler options that directly affect an inlining ratio, unearthing common misbeliefs, and the evaluation of security implications on ML-oriented applications. Our major findings indicate that function inlining can be exploited for malicious purposes, which requires paying attention when building ML-based models.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work was partially supported by the grants from Institute of Information & Communications Technology Planning & Evaluation (IITP), funded by the Korean government (MSIT; Ministry of Science and ICT): No. RS-2024-00337414, No. RS-2024-00437306, No. RS-2020-II201821, and No. RS-2022-II220688. Additional support was provided by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education of the Government of South Korea : No. RS-2025-02293072. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous systems," 2015, https://www.tensorflow.org/.

[2] A. Abusnaina, M. Abuhamad, H. Alasmary, A. Anwar, R. Jang, S. Salem, D. Nyang, and D. Mohaisen, "Dl-fhmc: Deep learning-based fine-grained hierarchical learning approach for robust malware classification," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[3] A. Abusnaina, A. Anwar, S. Alshamrani, A. Alabduljabbar, R. Jang, D. Nyang, and D. Mohaisen, "Systematically evaluating the robustness of ml-based iot malware detection systems," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022.

[4] T. Ahmed, P. Devanbu, and A. A. Sawant, "Learning to find usages of library functions in optimized binaries," *IEEE Transactions on Software Engineering*, 2021.

[5] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022.

[6] H. Alasmary, A. Anwar, J. Park, J. Choi, D. Nyang, and A. Mohaisen, "Graph-based comparison of iot and android malware," in *Proceedings of the International Conference on Computational Social Networks*, 2018.

[7] O. Alrawi, C. Lever, K. Valakuzhy, K. Snow, F. Monrose, M. Antonakakis *et al.*, "The circle of life: A large-scale study of the iot malware lifecycle," in *30th USENIX Security Symposium*, 2021.

[8] J. Alves-Foss and V. Venugopal, "The inconvenient truths of ground truth for binary analysis," in *Binary Analysis Research Workshop*, 2022.

[9] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th USENIX Security Symposium*, 2017.

[10] ARM, "Arm developer," 2023, https://developer.arm.com/documentation/100067/0608/armclang-Command-line-Options/-fno-inline-functions.

[11] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium*, 2014.

[12] E. Bendersky, "pyelftools: a pure-python library for parsing elf and dwarf," 2024, https://github.com/eliben/pyelftools.

[13] Binutils, "Gnu binutils 2.40," 2023, https://ftp.gnu.org/gnu/binutils/.

[14] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[15] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *31st USENIX Security Symposium*, 2022.

[16] Cisco-Talos, "Binary function similarity," 2024, https://github.com/Cisco-Talos/binary_function_similarity.

[17] Clang, "Options to emit optimization reports," https://clang.llvm.org/docs/UsersManual.html#options-to-emit-optimization-reports, 2024.

[18] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," 1997.

[19] D. D. I. F. Committee, "Dwarf debugging information format version 5." 2017, https://dwarfstd.org/dwarf5std.html.

[20] Coreutils, "Gnu coreutils," 2023, https://github.com/coreutils/coreutils.

[21] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti, "The tangled genealogy of iot malware," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020.

[22] L. Dall'Aglio, L. Binosi, M. Carminati, S. Zanero, and M. Polino, "Highliner: Enhancing binary analysis through nlp-based instruction-level detection of c++ inline functions," *ACM Transactions on Privacy and Security*, 2025.

[23] T. Damásio, V. Pacheco, F. Goes, F. Pereira, and R. Rocha, "Inlining for code size reduction," in *Proceedings of the 25th Brazilian Symposium on Programming Languages*, 2021.

[24] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the ACM on Programming Languages*, 2020.

[25] S.-L. Developers, "Scikit-learn: Machine learning in python," 2023, https://scikit-learn.org/stable/.

[26] Diffutils, "Gnu diffutils 3.8," 2023, https://ftp.gnu.org/gnu/diffutils/.

[27] F. Ding, "Iot malware: Lizkebab (gafgyt)," 2023, https://github.com/ifding/iot-malware/tree/master/lizkebab.

[28] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy*, 2019.

[29] S. Dong, O. Olivo, L. Zhang, and S. Khurshid, "Studying the influence of standard compiler optimizations on symbolic execution," in *2015 IEEE 26th International Symposium on Software Reliability Engineering*, 2015.

[30] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[31] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd USENIX Security Symposium*, 2014.

[32] S. Eschweiler, K. Yakdan, E. Gerhards-Padilla *et al.*, "Discovre: Efficient cross-architecture identification of bugs in binary code." in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[33] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[34] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[35] Findutils, "Gnu findutils," 2023, https://github.com/aixoss/findutils.

[36] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao, "Coda: An end-to-end neural program decompiler," *Advances in Neural Information Processing Systems*, 2019.

[37] J. Gamblin, "Mirai source code," 2023, source code: https://github.com/jgamblin/Mirai-Source-Code.

[38] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[39] T. Glek and J. Hubicka, "Optimizing real world applications with gcc link time optimization," *arXiv preprint arXiv:1010.2196*, 2010.

[40] GSL, "Gnu scientific library 2.7.0," https://github.com/ampl/gsl, 2023, accessed: 2024-07-10.

[41] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[42] Y. He, J. Lou, Z. Qin, and K. Ren, "Finer: Enhancing state-of-the-art classifiers with feature attribution to facilitate security analysis," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[43] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis," *ACM Transactions on Software Engineering and Methodology*, 2023.

[44] A. Jia, M. Fan, X. Xu, W. Jin, H. Wang, and T. Liu, "Cross-inlining binary function similarity detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024.

[45] J. Jiang, G. Li, M. Yu, G. Li, C. Liu, Z. Lv, B. Lv, and W. Huang, "Similarity of binaries across optimization levels and obfuscation," in

Proceedings of the European Symposium on Research in Computer Security, 2020.

[46] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[47] T. Johnson, M. Amini, and X. D. Li, "Thinlto: scalable and incremental lto," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization*, 2017.

[48] L. Jones, A. Sellers, and M. Carlisle, "Cardinal: similarity analysis to defeat malware compiler variations," in *2016 11th International Conference on Malicious and Unwanted Software*, 2016.

[49] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, 2018.

[50] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," *arXiv preprint arXiv:1905.08325*, 2019.

[51] W. Kenton, "Three-sigma limits definition," 2023, https://www.investopedia.com/terms/t/three-sigma-limits.asp.

[52] R. Kettlewell, "Inline functions in c," 2024, https://www.greenend.org.uk/rjk/tech/inline.html.

[53] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering*, 2022.

[54] ——, "Binary code similarity analysis (bcsa) tool," 2023, https://github.com/SoftSec-KAIST/TikNib/tree/master.

[55] H. Kim, J. Bak, K. Cho, and H. Koo, "A transformer-based function symbol name inference model from an assembly language for binary reversing," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023.

[56] T. Kim, A. Ding, S. Etigowni, P. Sun, J. Chen, L. Garcia, S. Zonouz, D. Xu, and D. Tian, "Reverse engineering and retrofitting robotic aerial vehicle control firmware using dispatch," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022.

[57] H. Koo, S. Park, and T. Kim, "A look back on a function identification problem," in *Proceedings of the 37th Annual Computer Security Applications Conference*, 2021.

[58] T. Le, T. V. Nguyen, T. Le, D. Phung, P. Montague, O. De Vel, and L. Qu, "Maximal divergence sequential auto-encoder for binary software vulnerability detection," in *International Conference on Learning Representations*, 2019.

[59] M. Learn, "Function inlining problems," 2021, https://learn.microsoft.com/en-us/cpp/error-messages/tool-errors/function-inlining-problems?view=msvc-170.

[60] T. Lei, J. Xue, Y. Wang, T. Baker, and Z. Niu, "An empirical study of problems and evaluation of iot malware classification label sources," *Journal of King Saud University-Computer and Information Sciences*, 2024.

[61] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[62] Lighttpd, "Lighttpd 2.0.0," 2023, https://download.lighttpd.net/lighttpd/snapshots-2.0.x/.

[63] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "Poster: Vulnerability discovery with function representation learning from unlabeled projects," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[64] W. Lin, Q. Guo, J. Yin, X. Zuo, R. Wang, and X. Gong, "Fsmell: Recognizing inline function in binary code," in *European Symposium on Research in Computer Security*, 2023.

[65] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: Cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[66] LLVM Project, "CGSCC Pass Manager.h," 2024, https://llvm.org/doxygen/CGSCCPassManager_8h.html.

[67] ——, "Early CSE Pass," 2024, https://llvm.org/doxygen/structllvm_1_1EarlyCSEPass.html.

[68] ——, "Llvm link time optimization: Design and implementation," 2024, https://llvm.org/docs/LinkTimeOptimization.html.

[69] ——, "LLVM Passes Documentation," 2024, https://llvm.org/docs/Passes.html.

[70] ——, "Simplify CFG Pass," 2024, https://llvm.org/doxygen/classllvm_1_1SimplifyCFGPass.html.

[71] ——, "SROA Pass," 2024, https://llvm.org/doxygen/classllvm_1_1SROAPass.html.

[72] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in Neural Information Processing Systems*, 2017.

[73] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *Proceedings of the Network and Distributed System Security Symposium*, 2023.

[74] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium*, 2022.

[75] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019.

[76] H. Mauelshagen, "Logical volume manager," 2023, https://github.com/lvmteam/lvm2/tags.

[77] Melissa, "Repository of malware samples," 2023, https://virusshare.com/.

[78] T. Micro, "Bash vulnerability (shellshock) exploit emerges in the wild, leads to bashlite malware," 2014, https://web.archive.org/web/20181129100545/https://blog.trendmicro.com/trendlabs-security-intelligence/bash-vulnerability-shellshock-exploit-emerges-in-the-wild-leads-to-flooder/.

[79] O. MPI, "Open mpi 4.1.5," 2023, https://www.open-mpi.org/software/ompi/v4.1/.

[80] Nginx, "Nginx," 2023, https://github.com/nginx.

[81] OpenSSL, "Openssl 3.1.4," 2023, https://github.com/openssl/openssl.

[82] Y. Park, H. Lee, J. Jung, H. Koo, and H. K. Kim, "Benzene: A practical root cause analysis system with an under-constrained state mutation," in *2024 IEEE Symposium on Security and Privacy*, 2024.

[83] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Probabilistic naming of functions in stripped binaries," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020.

[84] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," in *Proceedings of the Network and Distributed System Security Symposium*, 2021.

[85] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *IEEE Transactions on Software Engineering*, 2022.

[86] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[87] L. Project, "Llvm 14.0.0 release version," 2023, https://releases.llvm.org/14.0.0/docs/ReleaseNotes.html.

[88] ——, "Llvm language reference manual," 2024, https://llvm.org/docs/LangRef.html#intrinsic-functions.

[89] PuTTY, "Putty 0.79," 2023, https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html.

[90] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium*, 2015.

[91] M. Reddy, "Api design for c++," 2011, elsevier, 2011.

[92] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: An empirical study," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.

[93] X. Ren, M. Ren, Y. Lei, and J. Ming, "Revisiting optimization-resilience claims in binary diffing tools: Insights from llvm peephole optimization analysis," *Proceedings of the ACM on Software Engineering*, 2025.

[94] R. W. Scheifler, "An analysis of inline substitution for a structured programming language," *Communications of the ACM*, 1977.

[95] scikit learn, "Resample," 2023, https://scikit-learn.org/stable/modules/generated/sklearn.utils.resample.html.

[96] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2018.

[97] S. P. E. C. (SPEC), "Spec cpu2006 documentation," 2024, https://www.spec.org/cpu2006/docs/.

[98] T. Theodoridis, T. Grosser, and Z. Su, "Understanding and exploiting optimal function inlining," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[99] Valgrind, "Valgrind 3.21.0," 2023, https://github.com/rantoniello/valgrind.

[100] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "Jtrans: Jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[101] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th international conference on software engineering*, 2016.

[102] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: An image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[103] X. Xu, Q. Zheng, Z. Yan, M. Fan, A. Jia, Z. Zhou, H. Wang, and T. Liu, "Patchdiscovery: Patch presence test for identifying binary vulnerabilities based on key basic blocks," *IEEE Transactions on Software Engineering*, 2023.

[104] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[105] H. Xue, S. Sun, G. Venkataramani, and T. Lan, "Machine learning-based analysis of program binaries: A comprehensive study," *IEEE Access*, 2019.

[106] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *2016 IEEE Symposium on Security and Privacy*, 2016.

[107] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations." *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[108] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium*, 2018.

[109] Y. Zhang, M. Sirlanci, R. Wang, and Z. Lin, "When compiler optimizations meet symbolic execution: An empirical study," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[110] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

Appendix A
Supplementary Experiments and Analyses

A. *DWARF Information for Function Inlining*

TABLE VIII: Debugging information fields in DWARF [19] associated with function inlining.

Name	Type	Description
DW_TAG_inlined_subroutine	T	Particular inlined instance of a (sub)routine
DW_AT_abstract_origin	A	Pointer to the DIE of the inlined subprogram
DW_AT_inline	A	Constant describing inlining behavior
DW_AT_decl_file	A	File containing source declaration
DW_AT_low_pc	A	Low address of a machine code
DW_AT_high_pc	A	High address of a machine code
DW_AT_call_file	A	File containing an inlined subroutine call
DW_AT_call_line	A	Line number of an inlined subroutine call
DW_AT_call_column	A	Column position of an inlined subroutine call

B. *Code Semantics on Vulnerability Detection*

Table IX displays CVEs and corresponding vulnerable functions described in Section VI-A5. For example, with Trex [85], we discover that the similarity rank of `BN_dec2bn` has increased after being inlined into `BN_asc2bn`: *e.g.,* $1,044 \rightarrow 14$, which indicates that the model recognizes the code semantics. As such, identifying the semantics of a vulnerability within the inlining function (*i.e.,* containing a vulnerable function) is a good sign for detection. However, based on our observation, the mixture of different code semantics could degrade the model's performance by additional optimizations and nested-inlining may significantly.

TABLE IX: The specific names and CVEs of the vulnerable functions we explored in Section VI-A5.

	CVE	Query Function
Netgear R7000	**CVE-2016-2182**	`BN_bn2dec`
	CVE-2019-1563	`CMS_decrypt`
	CVE-2016-6303	`MDC2_Update`
	CVE-2019-1563	`PKCS7_dataDecode`
TP-Link Deco-M4	**CVE-2016-2182**	`BN_bn2dec`
	CVE-2016-0797	`BN_dec2bn`
	CVE-2016-0797	`BN_hex2bn`
	CVE-2019-1563	`CMS_decrypt`
	CVE-2016-2105	`EVP_EncodeUpdate`
	CVE-2019-1563	`PKCS7_dataDecode`
	CVE-2016-0798	`SRP_VBASE_get_by_user`
	CVE-2016-2176	`X509_NAME_oneline`

C. *Augmented Training with Inlined Functions*

Experiments and Results. We perform augmented training with inlined functions for a malware detection task (T3) as a potential defense. We generate new extreme-inlining variants and add them to the training dataset that consists of 159 benign samples, 50 Mirai and 25 Gafgyt samples. Note that we use a different extreme inlining recipe for augmentation (Table XI) to ensure unseen patterns for testing. The preliminary results in Table XII demonstrate two messages. First, the model's performance (*e.g.,* F1) increases in neural networks such as CNN and DNN after augmented training. Second, non-neural models, such as logistic regression and random forest, do not benefit from augmented training.

TABLE X: Descriptions of static features in an executable binary from Binkit [53]. The 36 features (1-36) depict instruction-level features, while the 20 (37-56) and six (57-62) features are relevant to a control flow graph and a call graph, respectively. We analyze how each static feature has been affected by extreme inlining.

Index	Feature Description
1	Number of instructions
2	Average number of instructions
3	Number of unknown instructions
4	Average number of unknown instructions
5	Number of absolute arithmetic instructions
6	Average number of absolute arithmetic instructions
7	Number of arithmetic instructions
8	Average number of arithmetic instructions
9	Number of comparison instructions
10	Average number of comparison instructions
11	Number of absolute control transfer instructions
12	Average number of absolute control transfer instructions
13	Number of conditional control transfer instructions
14	Average number of conditional control transfer instructions
15	Number of group jump instructions
16	Average number of group jump instructions
17	Number of absolute data transfer instructions
18	Average number of absolute data transfer instructions
19	Number of data transfer instructions
20	Average number of data transfer instructions
21	Number of control transfer instructions
22	Average number of control transfer instructions
23	Number of group call instructions
24	Average number of group call instructions
25	Number of group return instructions
26	Average number of group return instructions
27	Number of miscellaneous instructions
28	Average number of miscellaneous instructions
29	Number of shift instructions
30	Average number of shift instructions
31	Number of logic instructions
32	Average number of logic instructions
33	Number of bit flag instructions
34	Average number of bit flag instructions
35	Number of floating-point instructions
36	Average number of floating-point instructions
37	Size of CFG
38	Number of loops in a CFG
39	Number of interprocedural loops in a CFG
40	Number of strongly connected components in a CFG
41	Number of back edges in a CFG
42	Number of breadth-first search edges in a CFG
43	Maximum width of CFG
44	Maximum depth of CFG
45	Sum of the sizes of all loops in a CFG
46	Sum of the sizes of all interprocedural loops in a CFG
47	Sum of the sizes of all strongly connected components in a CFG
48	Average size of loops in a CFG
49	Average size of interprocedural loops in a CFG
50	Average size of strongly connected components in a CFG
51	Number of incoming edges in a CFG
52	Number of outgoing edges in a CFG
53	Total number of edges (in-degree + out-degree) in a CFG
54	Average number of incoming edges in a CFG
55	Average number of outgoing edges in a CFG
56	Average number of edges (in-degree + out-degree) in a CFG
57	Number of caller functions in a CG
58	Number of callee functions in a CG
59	Number of imported callee functions in a CG
60	Number of incoming calls in a CG
61	Number of outgoing calls in a CG
62	Number of imported calls in a CG

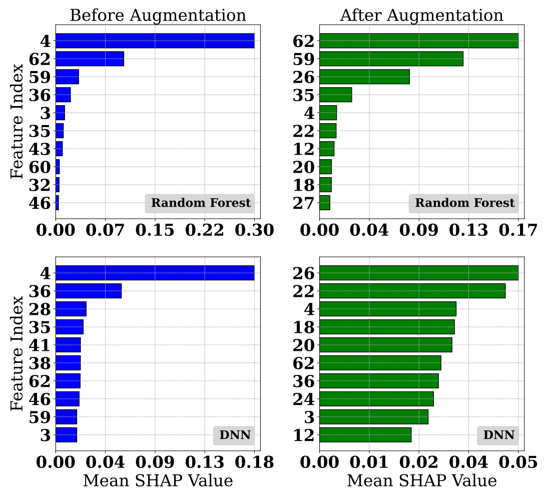


Fig. 13: The top 10 most influential features (Table X) based on mean absolute SHAP values for a traditional ML model (*e.g.,* random forest) and a neural network model (*e.g.,* DNN) before and after augmentation with extreme inlining. A higher SHAP score denotes a greater contribution of the feature to the model's decision. Although baseline feature rankings differ across models, augmentation reshapes feature importance more strongly DNN than random forest, suggesting that neural architectures adapt more readily to extreme-inlining.

In-depth Analysis with SHAP. To assess how augmentation alters model behavior, we apply SHapley Additive exPlanations (SHAP) [72] to measure feature contributions across ten iterations, comparing the mean absolute SHAP values before and after augmentation. It is noted that we attempt to understand how each feature contributes to a model decision, as Explainable AI has a fidelity issue: *i.e.,* explanations approximate a decision process, but they do not perfectly reflect it. Figure 13 depicts the Top 10 contributing features for two representative models: random forest from a non-neural network and DNN from a neural network. Extreme inlining impacts the entire feature space, modifying not only instruction frequencies but also control-flow and call-graph characteristics. Our findings show that, after augmented training, neural model (DNN) exhibit a more pronounced shift in feature importance compared to the traditional model (random forest) indicating that augmentation has a more substantial effect on neural architectures. Interestingly, across all models without augmentation training, the decisions are dominated by Feature 4 (*i.e.,* number of unknown instructions). Our further analysis reveals that `rep stosq, qword ptr [rdi], rax` frequently appears in malware samples, which likely accounts for the strong contribution of this feature to detection.

Appendix B
Artifact Appendix

A. Abstract

Function inlining is a common compiler optimization that replaces a function call with the function's body to improve performance, but it can also alter the structural properties of binaries used in machine learning–based security analysis. To

TABLE XI: Summary of extreme inlining recipes for each task (T1–T5).

Type	Package or Application(s)	Extreme Inlining Recipe Settings			
		T1–2, T5	T3	T4	T3* (augmentation variants)
Benign	coreutils	-Oz -inline-threshold=2225	-O3 -flto=full; -O3 -flto=full -inline-threshold=200000	-	-O3 -inline-threshold=2225
	binutils	-O3	-	-	-
	diffutils	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	findutils	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	openssl	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	lvm2	-O3 -flto=full	-	-	-
	gsl	-Oz -inline-threshold=2225	-O3 -flto=full	-	-O3 -inline-threshold=2225
	valgrind	-Oz -inline-threshold=2225	-	-	-O3 -inline-threshold=2225
	openmpi	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	putty	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	nginx	-Oz -inline-threshold=2225	-O3 -flto=full	-	-O3 -inline-threshold=2225
	lighttpd	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
Malware	Mirai	-	-O(1|2|3|s|z) –inline-threshold=(225|500225)		-O(1|2|3|s|z) –inline-threshold=(225|2000)
	Gafgyt	-	-O(1|2|3|s|z) –inline-threshold=(225|500225) -flto=full		-O(1|2|3|s|z) –inline-threshold=(225|2000) -flto=full

TABLE XII: Performance comparison on the malware detection task (T3) with newly generated extreme-inlining variants. Interestingly, augmenting the training set improves neural models (*e.g.,* CNN, DNN), while non-neural models (*e.g.,* logistic regression, random forest, CatBoost) show little benefit.

Training Setup	ML Model	Malware in the Wild				Malware with Extreme Inlining			
		Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
Before Augmentation	Logistic Regression	**0.99 ± 0.00**	**0.99 ± 0.00**	0.99 ± 0.01	**0.99 ± 0.00**	0.60 ± 0.02	0.64 ± 0.03	0.47 ± 0.05	0.54 ± 0.03
	Random Forest	**0.99 ± 0.00**	**0.99 ± 0.00**	**0.99 ± 0.00**	**0.99 ± 0.00**	0.78 ± 0.01	0.70 ± 0.02	0.95 ± 0.03	0.81 ± 0.02
	CatBoost	0.98 ± 0.00	**0.99 ± 0.00**	0.97 ± 0.01	0.98 ± 0.00	**0.78 ± 0.01**	0.70 ± 0.01	**0.97 ± 0.01**	**0.82 ± 0.01**
	KNN	**0.99 ± 0.00**	**0.99 ± 0.00**	0.99 ± 0.01	**0.99 ± 0.00**	0.68 ± 0.04	0.69 ± 0.03	0.65 ± 0.09	0.67 ± 0.06
	Average (Non-Neural)	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.71 ± 0.08	0.68 ± 0.03	0.76 ± 0.20	0.71 ± 0.11
	CNN	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	0.75 ± 0.05	**0.76 ± 0.09**	0.77 ± 0.10	0.76 ± 0.04
	DNN	**0.99 ± 0.00**	**0.99 ± 0.00**	**0.99 ± 0.00**	**0.99 ± 0.00**	0.71 ± 0.04	0.73 ± 0.03	0.67 ± 0.10	0.70 ± 0.06
	Average (Neural)	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	0.73 ± 0.03	0.75 ± 0.02	0.72 ± 0.05	0.73 ± 0.03
After Augmentation	Logistic Regression	0.96 ± 0.01	**0.99 ± 0.01**	0.94 ± 0.02	0.96 ± 0.01	0.64 ± 0.07	0.98 ± 0.02	0.28 ± 0.14	0.42 ± 0.17
	Random Forest	0.98 ± 0.01	**0.99 ± 0.01**	0.97 ± 0.01	0.98 ± 0.01	0.69 ± 0.02	0.90 ± 0.00	0.38 ± 0.03	0.50 ± 0.02
	CatBoost	0.98 ± 0.01	**0.99 ± 0.01**	0.96 ± 0.02	0.98 ± 0.01	**0.84 ± 0.02**	0.99 ± 0.01	**0.69 ± 0.03**	**0.81 ± 0.02**
	KNN	**0.99 ± 0.00**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.00**	0.81 ± 0.01	**0.99 ± 0.00**	0.62 ± 0.03	0.77 ± 0.02
	Average (Non-Neural)	0.98 ± 0.01	0.99 ± 0.00	0.97 ± 0.02	0.98 ± 0.01	0.75 ± 0.09	0.97 ± 0.05	0.49 ± 0.18	0.63 ± 0.18
	CNN	**0.99 ± 0.00**	**0.99 ± 0.01**	0.98 ± 0.01	**0.99 ± 0.00**	0.84 ± 0.02	**0.99 ± 0.00**	0.68 ± 0.05	0.81 ± 0.03
	DNN	**0.99 ± 0.00**	**0.99 ± 0.01**	0.98 ± 0.01	**0.99 ± 0.00**	0.81 ± 0.01	**0.99 ± 0.00**	0.62 ± 0.03	0.77 ± 0.02
	Average (Neural)	0.99 ± 0.00	0.99 ± 0.01	0.98 ± 0.00	0.99 ± 0.00	0.83 ± 0.02	0.99 ± 0.00	0.65 ± 0.04	0.79 ± 0.03

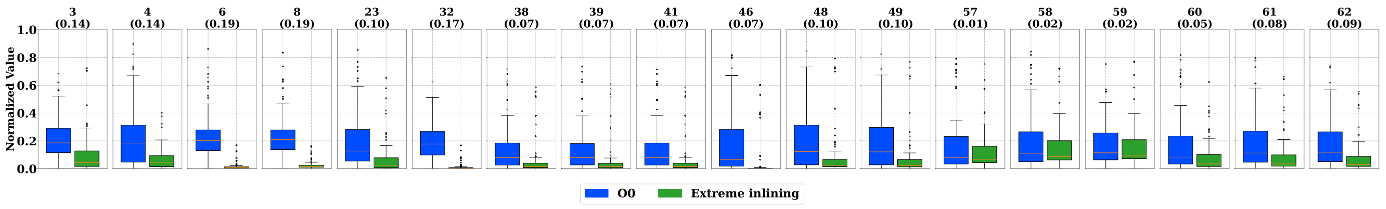


Fig. 14: Comparison of selective static features between little inlining (`-O0`) and extreme inlining (*e.g.,* compiler options: `-O3 -flto=full -inline-threshold=200000`). The features include instructions (Features $3, 4, 6, 8, 23, 32$), control flow graphs (Features $38, 39, 41, 46, 48, 49$), and call graphs (Features $57, 58, 59, 60, 61, 62$). The numbers on top denote the features and the gap between the two means (parenthesis) (Table X in Appendix). This example illustrates the significant gaps in the mean values, which can threaten the robustness of ML-based models. Note that we normalize all values for concise comparison. See Section VI-B4 in detail.

investigate this phenomenon, our study systematically examines the security implications of inlining on ML-based binary analysis. This artifact accompanies our paper and provides ready-to-use datasets and scripts for verifying the main results presented therein. It includes the source code and build scripts used to explore inlining-related compiler flags toward *extreme inlining*, modified feature extraction tools, runnable ML models, and analysis scripts for regenerating the main results reported in the paper. Together, these resources enable examiners to validate our findings and further examine the impact of extreme inlining on ML-based security tasks, including binary code similarity detection (T1), function name prediction

(T2), malware detection (T3), malware family prediction (T4), and vulnerability detection (T5).

B. Description & Requirements

The provided artifact includes the source code, build scripts, curated datasets, and Python analysis utilities used to reproduce the experimental results presented in our paper. The artifact is organized into four main components:

- **Dataset Construction.** Includes the source code and build scripts for exploring inlining-related compiler flags under diverse configurations toward *extreme inlining*. Although the full dataset construction process is provided for completeness, ready-to-use datasets are included to allow quick verification without the need for recompilation.
- **Feature Extraction.** Contains a modified version of *TikNib*, configured to extract static and semantic features used across Tasks T3 and T4 in the study.
- **ML Models.** Provides runnable ML models and associated datasets used for evaluating robustness under extreme inlining, across five binary analysis security tasks (T1–T5) and 18 representative models.
- **Main Results and Plots.** Includes CSV result files and Python scripts to regenerate the primary analysis results reported in the paper. A verification script is also provided to reproduce these results directly from the included datasets.

Scope. This artifact focuses on verifying the reported findings rather than regenerating datasets from scratch. For T3 and T4, due to the stochastic nature of ML models and Monte Carlo cross-validation, exact results may vary slightly from those reported in the paper; however, the overall trends remain consistent.

Security, privacy, and ethical concerns. The provided scripts operate entirely on open-source data and do not perform any destructive or privacy-sensitive actions. Binaries generated from the proprietary SPEC CPU2006 benchmark are not shared due to licensing restrictions. Therefore, no security, privacy, or ethical concerns apply.

1) How to access: The artifacts are available open-source on Zenodo [8].

2) Hardware dependencies: Experiments were conducted on a workstation running Ubuntu 20.04 (64-bit) with an Intel Xeon Gold 5218R @ 3.00 GHz CPU, 512 GB RAM, and two NVIDIA RTX A6000 GPUs.

3) Software dependencies: The artifact requires the following software environment: LLVM/Clang 14.0.0, Python 3.8 or newer (with `pip` $\geq$ 23.0 and `Conda` $\geq$ 23.7.4), Docker 20.10.22 or newer, and IDA Pro 8.2 or 8.3.

4) Benchmarks: The experiments rely on multiple datasets (See Table V) and benchmark implementations used across the evaluated ML-assisted binary analysis tasks. Specifically, we use the BCSD benchmark [74] for T1 and T5, including models such as Asm2Vec, Gemini, Trex, SAFE, and BinShot. For T2, we retrain AsmDepictor [55] using our LLVM-compiled dataset. For the malware-related tasks (T3 and T4), we curated datasets from different sources. For T3, we used samples collected from VirusShare [77]. For T4, we curated the dataset from Alrawi *et al.* [7], whose artifacts are publicly available at [9]. The corresponding models include four traditional ML classifiers built with Scikit-learn [25] and deep learning models re-implemented from Abusnaina *et al.* [3]. For T5, we use the BCSD benchmark along with publicly available vulnerable firmware corpora [16].

[8]https://doi.org/10.5281/zenodo.17759528

[9]https://badthings.info/#bins

C. Artifact Installation & Configuration

1) Installation: To install the artifact, download the repository from the provided link and navigate to the project root directory. All required dependencies are listed in Section B-B and can be installed using standard package management tools such as `pip` or `conda`. Each directory is self-contained and includes a `START_EVALUATION.md` file and configuration scripts to guide installation and execution. Note that a pre-configured Docker image is also provided, alternatively.

For tasks T1, T2, and T5, create the provided Conda environment:

```
cd ml-model/t1_t2_t5
conda env create -f environment.yml
conda activate inline_ae
```

For tasks T3 and T4, using the Conda environment is optional; dependencies are available in the root directory:

```
pip install -r requirements.txt
```

2) Basic Test: To verify that the artifact is correctly installed and functional, execute any of the provided task scripts. For example for T1:

```
cd ml-model/t1_t2_t5/T1_bcsd
bash run.sh
```

D. Experiment Workflow

The artifact is organized into modular components, each corresponding to a stage in the experimental workflow. The overall process can be summarized as follows: (i) dataset construction under diverse inlining configurations using the provided compilation sweep scripts; (ii) feature extraction with the modified `TikNib` framework [54]; (iii) training and evaluation of machine learning models across five security tasks (T1–T5); and (iv) analysis and visualization of the impact of inlining on binaries and their associated statistical features, which can be regenerated using the provided analysis scripts.

E. Major Claims

- (C1): The artifact provides the complete source code, build scripts, and curated datasets used to evaluate the impact of function inlining under diverse compiler configurations. These resources enable transparent verification of the dataset generation process and reproducibility of the experimental setup described in Section VI.
- (C2): The provided ML models and feature extraction tools reproduce all key evaluations across the five ML-based security tasks (T1–T5), supporting the paper's

findings on the sensitivity of model robustness to extreme inlining, as discussed in Section VI-A.

- (C3): The included analysis scripts regenerate all primary quantitative results and plots (Figures 7–14), illustrating the inlining trends across configurations and the impact of extreme inlining on statistical features compared to the non-inlined baseline, as detailed in Section VI-B.

F. Evaluation

- (E1): **Dataset Construction and Compilation Sweep** [15 person-minutes + several compute-hours] Run the provided dataset construction scripts to reproduce the compilation sweep under diverse inlining configurations. The build framework systematically explores optimization and hidden inlining flags defined in `config.yaml`. Depending on the selected projects and increments, each dataset build may take several hours on a multi-core system. For validation purposes, reviewers may execute a reduced configuration (*e.g.,* a limited subset of projects or thresholds) to confirm functionality without requiring full-scale recompilation.
- (E2): **Feature Extraction and Visualization** [10 person-minutes + moderate compute time] Execute the modified `TikNib` pipeline to extract binary-level features and generate summary statistics or visualizations. This step produces the feature representations used in training T3–T4 and feature distortions (Figures 14).
- (E3): **Model Evaluation Across Tasks** [10 person-minutes + compute time depending on task] Evaluate the robustness of the ML models against extreme inlining using the provided scripts in the `ml-model/` directory. Approximate running times per task are as follows: T1 and T5 — approximately 2 hours, T2 — about 1 hour, T3 — around 10 minutes, and T4 — about 20 minutes.

Successful completion of Experiments (E1)–(E3) reproduces the main findings of the paper, validating the reproducibility and completeness of the provided artifact.

G. Notes on Reusability

The provided artifact is designed for reuse and extension in future research on extreme inlining, binary analysis, and ml–based security. Each component—dataset construction, feature extraction, model evaluation, and analysis—is modular and independently executable. By adjusting the configuration files (*e.g.,*, `config.yaml`) and following the detailed instructions in the accompanying `README.md` files, users can reproduce our experiments or adapt them to new datasets, compilers, or model architectures. For convenience, we also provide a prebuilt Docker image for reproducing the main experiments. As a final note, the modular design also allows researchers to explore more extreme inlining degrees could be explored automatically through our tuning strategy, providing an avenue for future work on adversarial code transformation.